Complete Turbo Pascal

Third Edition

Covers Version 5.0

Scott Foresman IBM Computer Books

Jeff Duntemann

Third Edition

Complete Turbo Pascal

Third Edition

Complete Turbo Pascal

Jeff Duntemann

Scott, Foresman and Company
Glenview, Illinois London

Library of Congress Cataloging-in-Publication Data

Duntemann, Jeff.
Complete Turbo Pascal / Jeff Duntemann.—3rd ed.
p. cm.
"Covers version 5.0"—Cover.
Includes index.
ISBN 0-673-38355-5
1. Pascal (Computer program language) 2. Turbo Pascal (Computer program) I. Title.
QA76.73.P2D86 1989
005.13'3—dc19 88-30821
CIP

2 3 4 5 6 RRC 93 92 91 90 89 88

ISBN 0-673-38355-5

Notice of Liability

The information in this book is distributed on an "As Is" basis, without warranty. Neither the author nor Scott, Foresman and Company shall have any liability to customer or any other person or entity with respect to any liability, loss, or damage caused or alleged to be caused directly or indirectly by the programs contained herein. This includes, but is not limited to, interruption of service, loss of data, loss of business or anticipatory profits, or consequential damages from the use of the programs.

Scott, Foresman professional books are available for bulk sales at quantity discounts. For information, please contact the Marketing Manager, Professional Books Group, Scott, Foresman and Company, 1900 East Lake Avenue, Glenview, IL 60025.

For Carol,
who builds structures of love

Foreword to the Third Edition

It was January 1984, and I was bundled up in my sweater, beating away on a cranky Xerox 820-II CP/M machine in the chilly bowels of Xerox MIS/DP in Rochester, NY. My phone rang. It was Dave Smereski, a hacker friend of mine in another building, and he was *excited.*

"Jeff, Jeff, you gotta see this. I mean, you have GOTTA see this. . ."

"This what?"

"It's a Pascal compiler. But I mean, you have *gotta* see this. . . "

"Dave, I have three Pascal compilers."

"But it only costs $49.95! And you have *gotta* see it. . ."

$49.95? Xerox had paid $350 for its copy of Pascal/MT+ for the 820. I had paid about that much for my copy of IBM PC Pascal on which I worked at home. And that *third* Pascal compiler. . .

"Dave, I will *not* waste $anything.95 again on a cheap Pascal compiler."

Smereski took another deep breath. "It's not like JRT, really. I've never seen anything like it. It's got a built-in editor. It's folded into the fourth dimension. It compiles fifty lines of code per *second*. . ."

My eyebrows rose. I was at that time in the seventh minute of a 10-minute compile on Pascal/MT+, after which would come a 3-minute link step, after which would come any number of peculiar bugs, and the whole thing would start again.

"Gimme a break."

"Be that way," Smereski said, and hung up.

Ten minutes later he roared in, earmuffs flapping, and thrust an 8" disk in my hands. I stuck it into the 820, executed a file called TURBO.COM, and nothing was ever the same again.

I had recently begun writing a book called *Pascal from Square One,* which took the (then) remarkable step of focusing on one individual Pascal compiler, Pascal/MT+. Over the first half of 1984, as I finished up the book, the tumult raised by Turbo Pascal showed no sign of abating, and little by little Pascal/MT+ slipped into total eclipse. By the time I was ready to turn in the book in August, Turbo Pascal was in the saddle, and I had written a book about a corpse.

My editor at Scott, Foresman was no fool. He suggested that I could collect the remainder of my advance and *maybe* see a published book someday, or I could take the manuscript back and artfully replace the MT+ specific material with comparable information on Turbo Pascal.

Turning *Pascal from Square One* into *Complete Turbo Pascal* took the rest of the year, in part because I left Xerox late that year to become Technical Editor of *PC Tech Journal* in Baltimore. The finished manuscript was not out the door until February of 1985, and as luck would have it, in March Borland International announced Turbo Pascal V3.0, with lots of new features. We pondered rewriting the book yet again, but thought the better of it: with some luck we would have the first book in print to focus on Borland's maverick compiler, which by that time had sold into close to a quarter million hands.

As it happened, Steve Wood beat us into print by a few weeks with his book on Turbo, but no matter: by the end of 1985, *Complete Turbo Pascal* was selling three or four thousand copies per month.

In late 1985 I started work on an advanced topics book called *Turbo Pascal Solutions*, but *Complete Turbo Pascal* was selling so well that Scott, Foresman suggested a second edition to bring the book up to date in line with Turbo Pascal V3.0. That took the early part of 1986, and in October *Complete Turbo Pascal, Second Edition* appeared, fatter and bluer and much more complete.

The revised second edition sold briskly for well over a year. I had in the meantime left *PC Tech Journal* to finish *Turbo Pascal Solutions* and become a full-time writer. That didn't last long—in the spring of 1987 I moved cross-continent to help Philippe Kahn found *Turbo Technix Magazine*, a journal devoted to Borland's programming language products.

There are advantages to being an insider. All that summer and fall I watched in fascination as the alpha test Turbo Pascal 4.0 took form in Borland R&D. Turbo Pascal had evolved through three versions, but the break after 3.0 was clean—the product was rewritten utterly from scratch. All the things we had longed for since 1983 were there: Separate compilation, large memory model, long integers, project management, and a host of other features I had never seen in *any* Pascal compiler.

So in August 1987 I formatted a pair of new Bernoulli cartridges and sat down to do it again. Editing a magazine is engrossing work, and by the time I was getting toward finished, release 5.0 of the compiler was in alpha test. That was not as tremendous a transition as the one from 3.0 to 4.0, but it was important nonetheless. I took the time to incorporate 5.0 features into the manuscript, even though it meant that close to a year—and 1500 sheets of paper—had to go into the final product.

The results are in your hands. I regret a little having to pull the references to CP/M-80, but Turbo Pascal has grown beyond the Z80. Turbo is not as simple a product as it once was, as the size of this Third Edition will attest. But the world is not as simple, either—640K RAM is now *de riguer*, and OS/2 is looming on the horizon. We've got some serious hacking to do.

Fifty lines per second? *Watch our dust.*

And thanks, Dave. Really.

Introduction: This Is Square One

You have it in your head to learn Pascal. To that end, you have this book in your hand. In front of you is your personal computer. It can be most any personal computer running the PC/MS operating systems. The book's specific examples address the IBM Personal Computer or a close compatible using PC DOS.

I'll have to assume that you have learned the basics of operating your computer. This includes how to boot up the system, how to format diskettes, how to interprét a directory listing, how to copy files from one diskette to another, and so on. I'll assume you understand the importance of making backup copies of anything which has any value to you. *Anything that you don't back up you will eventually lose.*

As I mentioned earlier, this book is about a specific Pascal compiler: Turbo Pascal from Borland International. So finally, I will assume that you own Turbo Pascal 4.0 or 5.0 for your computer.

Given those few assumptions, this book and your own common sense can teach you how to program in Pascal.

ACKNOWLEDGMENTS

Many thanks to the R&D crew at Borland International, and to the hundreds of people whose letters have contributed in small but important ways to the evolution of this book.

—*Jeff Duntemann*
Scotts Valley, California

Contents

Part One

The Idea of Pascal

1 The Pascal Process

1.1: PROGRAMS ARE BLUEPRINTS FOR ACTION

That eccentric wizard Ted Nelson, author of *Computer Lib* and guru of the amazing Xanadu Hypertext system, defines a computer simply as "a box that follows a plan." And way down deep, that's all it is: The computer is a box that represents numbers, letters, and words as electronic symbols, and it understands a limited repertoire of instructions for manipulating those symbols. The plan is a sequence of those instructions, arranged so that the computer, in following the sequence of instructions, will get something done.

The instructions themselves are almost ridiculously minute. Translated out of their jargon, computer instructions sound something like this:

```
Put the number 6 into Box A.

Put the number stored at memory location X into Box B.

Add Box A to Box B.  Put the result into Box B.

Store the contents of Box B at Memory Location Y.

If the contents of Box B is 0, perform Instruction #Z

instead of the next instruction.
```

As you might imagine, its takes thousands of these little instructions to do anything useful. Because computers are so fast, they can, in fact, do thousands of these instructions in a second and make it look as though they are executing a small number of large steps; in fact, they are executing an enormous number of *very* small steps. This sequence of steps, so carefully arranged and followed, is a computer program. It is the computer's action plan. You write it. The computer follows it.

High-Level Computer Languages

The process of writing programs using the minuscule instructions paraphrased above is called "assembly language programming." It is done when necessary, for various reasons. It should not be done unless it *must* be done. There are infinitely easier ways to tell computers what to do.

The most important is the use of "high-level" computer languages, like Pascal, BASIC, FORTRAN, COBOL, and such. In a nutshell, such languages cobble together big program steps out of many little program steps. They allow you to write programs by arranging the big program steps and not worrying about writing hordes of tiny program steps. For example, the simple action

```
Beep
```

toots a tone on your computer's speaker. The actual sequence of instructions for making this tone might consist of dozens of tiny instructions. But you don't care about that; if you can write the word **Beep** and let someone else worry about setting up those dozens of tiny instructions, so much the better.

That "someone else" is a high-level computer language. Pascal is such a language. Pascal allows you to write a series of large instructions like this

```
Remainder := Remainder - 1;
IF Remainder = 0 THEN
BEGIN
  Beep;
  Write('Warning!  Your time has run out!');
END;
```

in perhaps 15 seconds. The sequence of actual "machine" instructions to do these things may be 100 instructions long. No problem. Pascal does the converting. It arranges the machine instructions automatically while you run out to the kitchen for another bagel and prune juice.

Converting a series of large, English-language steps into a series of tiny (usually incomprehensible) machine instructions is what Pascal does. BASIC too. And COBOL, as well as all the others. The idea is the same, no matter how it happens to be expressed.

1.2: PROGRAMS THAT MAKE OTHER PROGRAMS

What we call Pascal actually is itself a computer program. Like all other entities that are run or manipulated on a computer, it is stored on a disk of some kind. You run it as you would run any other computer program.

When you program in Pascal, what you actually do is edit a text file (a file of human-readable words) and store it on a disk. The computer itself can do nothing with this text file; to the computer, it is just a jumble of letters, numbers, and punctuation marks:

```
PROGRAM Genesis;

BEGIN
  Writeln('In the beginning God created Adam--');
  Writeln('--THEN She brought out Version 2.0.')
END.
```

The Pascal compiler program, however, understands this rather absurd little program very well. The compiler program reads the text file from disk and looks at every word and symbol. Based on which words they are, and what order they are in, it builds a stream of those minute machine instructions and stores the stream as a file on disk. This file becomes a real computer program, which can be loaded into your computer and run. When run, it will print these words on your CRT screen:

```
In the beginning God created Adam--
--THEN She brought out version 2.0.
```

and quit.

So, in fact, the Pascal compiler is a computer program that creates other computer programs. The words *you* write (your program) are actually instructions to the Pascal compiler and *not* to the computer itself.

1.3: SETTING UP YOUR COMPILER

Most of what you've read so far applies to any Pascal compiler. The focus of this book is on a specific compiler, Turbo Pascal 4.0 and 5.0 from Borland International. From now on, assume that what is said applies to Turbo Pascal, although the broader concepts of Pascal go beyond any individual implementation of the language.

Earlier versions of Turbo Pascal could run comfortably in a 128K or even a 64K system. This is no longer the case. A practical minimum for doing anything at all is 512K, and for any serious work you'll want 640K.

Unlike a lot of language development systems, Turbo Pascal is still compact enough to be used effectively on diskettes. On *single* diskette systems, that may or may not be the case, as the essential Turbo Pascal files occupy more than half of a single 360K diskette. Certainly, a hard disk is recommended.

Setting Up on Diskettes

If you're going to work on diskettes, I recommend that you prepare two work disks for learning and using Turbo Pascal. One disk (which I will call your compiler disk) should contain the compiler and a few other files that the compiler needs to operate. This disk will reside in disk drive A:.

The other disk (which I will call your program disk) will reside in drive B:, and will contain your program source files, "include" files (to be explained later), and compiled, runnable object files. You will, in time, have a number of program disks for drive B:. It makes a good deal of sense to confine each major programming project to its own disk. With diskettes now available for as little as 25 cents each in quantity (get together with your friends and buy them by the hundred!), the cost of one disk per project is negligible compared to the confusion you will avoid.

You've heard this before, I'm sure, but it's important: The distribution disks that come from Borland International should be considered *read-only*. Under *no* circumstances should you try to write to a distribution master disk, and you should read from it as infrequently as possible. Once you've made your compiler working disk, put the master in a plastic diskette case at the bottom of the bureau drawer that contains your stamp collection and last year's Sears catalogs.

Not all the files on the distribution disks are necessary for you to work with Turbo Pascal. In addition to the compiler itself, there are several example programs written in Turbo Pascal and a READ ME file that contains latebreaking notes on your release of the compiler.

At *minimum*, you must copy the following files from the distribution master to your compiler work disk:

TURBO.EXE	The environment-based compiler program.
TURBO.TPL	The runtime library unit. Much of the machinery of the language is stored here and is linked into your application when you compile.

The first time you set up your disks, you might also want to copy the TINST.EXE file to your compiler disk. This is the installation program that lets you modify screen colors, keyboard assignments, and other facets of the programming environment. Once you've installed the details of the program to your satisfaction, you can delete TINST.EXE from your compiler disk.

Setting Up on a Hard Disk

Far better still is to use Turbo Pascal in connection with a hard disk. Space becomes a non-issue (unless, like too many others, your hard disk is 96 percent full all the time), and the speed with which files are read from disk is considerably faster.

Create a subdirectory for the compiler (mine is called TURBO), and simply copy all the files from all the distribution disks to the subdirectory:

```
C:\>MD TURBO
C:\>CD TURBO
C:\>COPY A:*.*     (Repeat for each distribution diskette)
```

1.4: TURBO PASCAL AS A PROGRAMMING ENVIRONMENT

"Environment" is a word I will be using frequently to describe the Turbo Pascal system. I call it the Environment because it is more than just a Pascal compiler. It is a place to get your work done.

Traditionally, a Pascal compiler is a separate program in itself. To develop programs in Pascal, you must load and run a text editor to edit your text. Then, you must exit the text editor and load and run the Pascal compiler to compile the program text you have just edited. After that (in most cases), you must load and run a linker program that prepares the compiled object code for running. Once you exit the linker program, then, and only then, can you load and run your compiled program to see if it works correctly.

This option is available to you, by the way, beginning with Turbo Pascal release 4.0. A command-line version of the compiler called TPC.EXE is shipped along with the Environment-based compiler. I won't have much to say about TPC.EXE until Part Three. It's useful mostly for enormous development projects involving dozens or perhaps even hundreds of separate source files.

Using a command-line compiler is a lot of loading and jumping into and out of programs. It takes a fair amount of time to move all that information to and from a hard disk, much less a diskette. The Turbo Pascal Environment, by contrast, is *one* program that loads once from disk and lets you do *everything* from inside it. What we call Turbo Pascal includes an editor, a Pascal compiler, a means of running compiled programs, and some simple debugging utilities, all in one program.

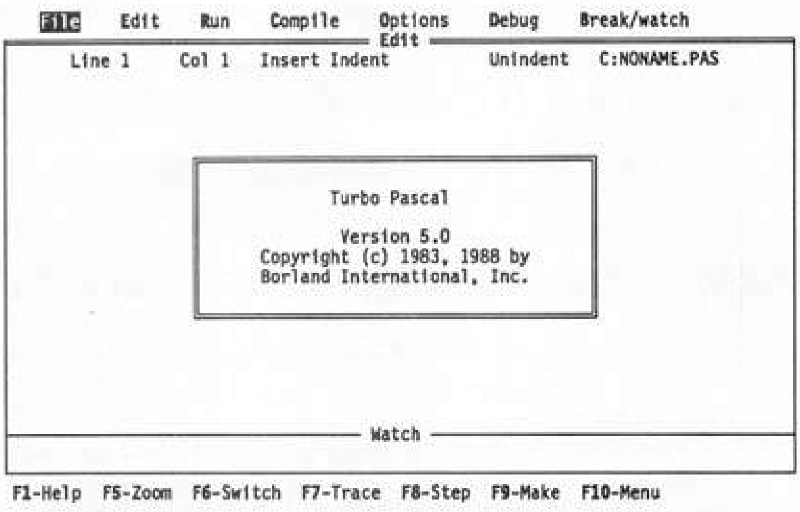
You run Turbo Pascal by typing its name:

```
A>TURBO
```

The screen will clear and the Turbo Pascal Environment will appear, as shown in Figure 1.1. When you press any key, the "Hello" box in the middle of the screen will

Figure 1.1

The Turbo Pascal "Hello" Screen


File Edit Run Compile Options Debug Break/watch
Edit
Line 1 Col 1 Insert Indent Unindent C:NONAME.PAS
Watch
F1-Help F5-Zoom F6-Switch F7-Trace F8-Step F9-Make F10-Menu

Turbo Pascal

Version 5.0
Copyright (c) 1983, 1988 by
Borland International, Inc.

disappear. What remains are two windows, the Edit window and the Run window (for Turbo Pascal 4.0) or the Watch window (for Turbo Pascal 5.0). These two windows are actually complete 25 × 80 "virtual" screens, each of which may be "zoomed" to occupy the entire visible display without disturbing the temporarily invisible screen.

The Edit window is the window in which you enter and modify Pascal programs. The Run window is the window in which the output from your programs (i.e., the words and numbers displayed with the **Write** and **Writeln** statements) appears. The Watch window allows you to inspect the values of variables while your program runs, through Turbo Pascal 5.0's Integrated Debugger (see Chapter 30).

1.5: EDITING, COMPILING, AND RUNNING A SIMPLE PROGRAM

The best way to learn anything is by doing, and you can get some Turbo Pascal practice by typing in some of the small programs from this book and then compiling and running them.

Part Three of *Complete Turbo Pascal* is a reference manual for using Turbo Pascal. It contains most of what you will need to know to run the compiler and editor, from elementary practice programs up to large, complicated programs with multiple units and nested include files. Because a beginner could easily drown in Part Three's ocean of technical details, this short section will describe the fundamentals of editing, compiling, and running small, simple programs such as the examples given in this book.

As you gain expertise in Pascal, you might spend some time reading through Part Three. It's not the sort of thing you'd want to curl up with on a rainy night, but when you must refer to some minuscule, arcane detail, it will help to have reviewed Part Three in advance.

Entering a Program

At the top of the Environment screen is a bar with several words on it: **F**ile, **E**dit, **R**un, **C**ompile, and **O**ptions. If you're using version 5.0, there will be two additional words: **D**ebug and **B**reak/watch. This bar is the menu bar, because these words are the "entry points" to menus containing further commands. (We'll explore some of them shortly and explain all of them in detail in Part Three.) One of the menu bar words is, however, a simple command: Edit. All you need do to begin entering a program is type the letter E.

Notice the flashing cursor that suddenly appears in the Edit window. This indicates that the Turbo Pascal Editor is ready to accept the characters that will make up your program.

Type the lines of the following short program, pressing Enter after each line:

```
PROGRAM Genesis;

BEGIN
  Writeln('In the beginning God created Adam--')
  Writeln('--THEN She brought out Version 2.0.')
END.
```

(Yes, there is a bug in the preceding program. Be a sport and type it in verbatim, just to see what happens. . . .)

If you make a mistake in typing, you can use the arrow keys to move the flashing cursor back to your misplaced characters and delete them or add what's missing.

There are quite a number of editor commands that you can use to build your program file with the Turbo Pascal editor. These include commands to search for certain patterns in the text, replace patterns with other patterns, delete or move blocks of text, read in additional text files from disk, and so on. These will be explained in detail in Chapter 26. For now, enter the example program as best you can.

Finally, you will decide that your program is finished and ready to be compiled. To exit the editor, hold down the CTRL key, then (while holding it) press (one at a time) K and D. The flashing cursor will disappear, indicating that you are no longer in the Edit window.

Saving Your Program to Disk

Your source program is still in memory. If you were to turn off the computer right now, you would lose it for good, because you have not yet saved it to disk. To save your program, you will need to bring up the Files menu.

There are two ways to do this. If one of the commands in the menu bar is highlighted, you can move the highlighted area back and forth from one command word to the other with the horizontal arrow keys. Move the highlighting to the **Files** command at the left end of the menu bar, and press return.

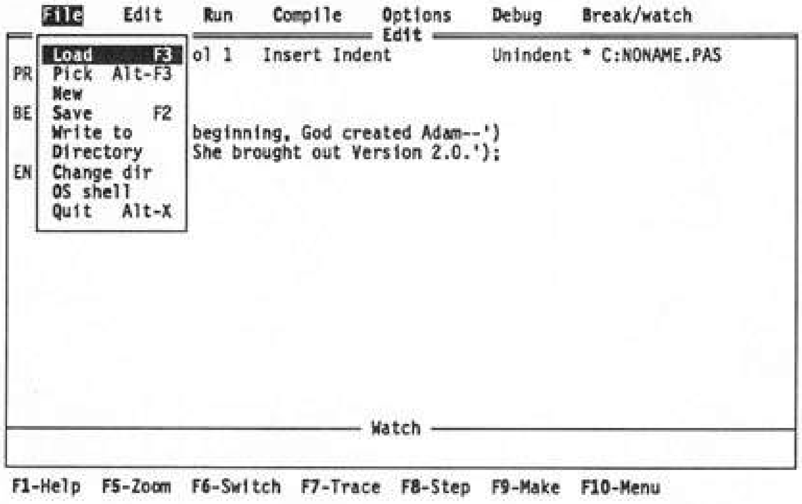
The other method is simpler: Just press the letter F, and the Files menu will appear instantly beneath the word Files (Figure 1.2). The working of the **Files** menu is very much like that of the menu bar across the top of the screen, but arranged vertically. If you press the up or down arrows, a highlighting bar will move up and down the Files menu, highlighting the different commands within the menu. Picking a command, again, can be done two ways: One way is to move the highlighting bar up and down with the arrow keys until it highlights the command you wish to select, then press Enter. The other, more direct, way is simply to press the first letter in the command.

All the menus in the Turbo Pascal Environment work this way. Keep that in mind as you explore the other commands in the main menu bar.

The simplest way to save your file is just to press the letter S, for Save. Instantly, a smaller box will appear, partly overlapping the **Files** menu (Figure 1.3). This is called a "dialog box," and there are quite a few of them lurking within the Environment.

Figure 1.2

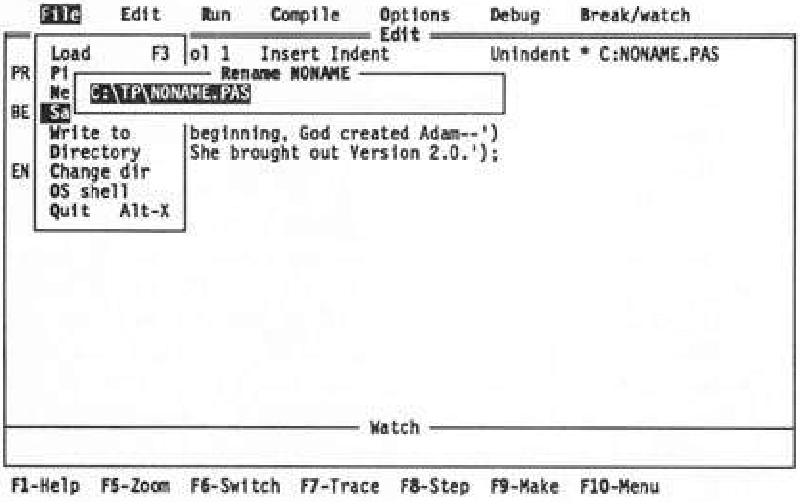
The Turbo Pascal Files Menu



```
  File    Edit    Run    Compile    Options    Debug    Break/watch
 ┌──────────────────────────────── Edit ───────────────────────────────┐
 │ Load      F3 │ol 1   Insert Indent        Unindent * C:NONAME.PAS
PR│ Pick   Alt-F3│
 │ New          │
BE│ Save      F2 │
 │ Write to     │beginning, God created Adam--')
 │ Directory    │She brought out Version 2.0.');
EN│ Change dir   │
 │ OS shell     │
 │ Quit   Alt-X │
 │
 ├──────────────────────────────── Watch ──────────────────────────────┤
 │
 └─────────────────────────────────────────────────────────────────────┘
 F1-Help  F5-Zoom  F6-Switch  F7-Trace  F8-Step  F9-Make  F10-Menu
```

Figure 1.3

The "Rename NONAME" Dialog Box



```
  File    Edit    Run    Compile    Options    Debug    Break/watch
 ┌──────────────────────────────── Edit ───────────────────────────────┐
 │ Load      F3 │ol 1   Insert Indent        Unindent * C:NONAME.PAS
PR│ Pi ┌────────── Rename NONAME ──────────┐
 │ Ne │ C:\TP\NONAME.PAS                  │
BE│ Sa └───────────────────────────────────┘
 │ Write to     │beginning, God created Adam--')
 │ Directory    │She brought out Version 2.0.');
EN│ Change dir   │
 │ OS shell     │
 │ Quit   Alt-X │
 │
 ├──────────────────────────────── Watch ──────────────────────────────┤
 │
 └─────────────────────────────────────────────────────────────────────┘
 F1-Help  F5-Zoom  F6-Switch  F7-Trace  F8-Step  F9-Make  F10-Menu
```

A dialog box will appear whenever the Environment needs input from you, most often to enter file names or paths. This particular dialog box appears because you need to give a name to the little program you just typed into the Edit Window. When you entered the Edit Window, the Environment gave a "default" name to the program you began typing in: NONAME.PAS. In fact, if you pressed Enter again at this point, the Environment would obediently save your program as a disk file with the name NONAME.PAS.

NONAME.PAS, however, is a pretty pointless name for anything. So type the name GENESIS.PAS in its place. You don't have to delete or backspace over NONAME.PAS; the first key you touch at this point will make NONAME.PAS vanish from the box. As soon as you finish entering your file name and press Enter again, your program will be saved as a disk file named GENESIS.PAS.

Compiling

Your program has been saved to disk and is ready to compile. The Files menu is still visible, however. To compile, you need to get to the Compile menu. Once again (and as almost always in the Environment) there are two ways to get there. The first, as you might expect by this time, is to press the right arrow key to move the highlight on the main menu bar to the Compile menu. Pressing the right arrow three times will do the job.

However, there's an easier way. Hold down the ALT key and press the letter C. The Files menu will vanish and the Compile menu will appear instantly under the Compile command on the main menu bar.

A pattern is emerging here. There are at least two paths to almost everything within the Environment: One is to use the arrow keys and the Enter key to highlight what you want and then pick it. This is sometimes called the "point and shoot" metaphor and is very handy when you're brand new to the Environment and don't yet know your way around. The other way is more direct, but less obvious: ALT plus the first letter in a main menu bar command will take you to that command. Within a menu like the Files or Compile menu, just touching the first letter of a command will execute that command.

The idea is to allow you to use fewer keystrokes to get things done as you become more familiar with the Turbo Pascal Environment without burning the bridge you needed to take as a beginner to move from one menu to another.

Right now, let's get into the heart of the Pascal process and compile GENESIS.PAS. Press C, the first letter of the Compile command within the Compile menu. A box will appear and will show you the progress of Turbo Pascal as it compiles your program. This box will display the line numbers of the program lines being compiled. For small programs (like GENESIS.PAS), you won't see the line numbers progress; you will simply see the word "Success" appear in the lower left corner of the box.

That's how fast Turbo Pascal is.

Now if, during the compile, Turbo Pascal detects an error in your program, it will display the error and its error number on the screen. In fact, if you typed in program

Genesis *exactly* as it was given earlier, you will see such a message when you try to compile it. A bright bar will appear on the top line of the Edit Window (it will be in flaming red if you have a color screen) with this message:

```
Error 85: ";" expected
```

The flashing cursor will be flashing under the W in the second **Writeln** word (Figure 1.4).

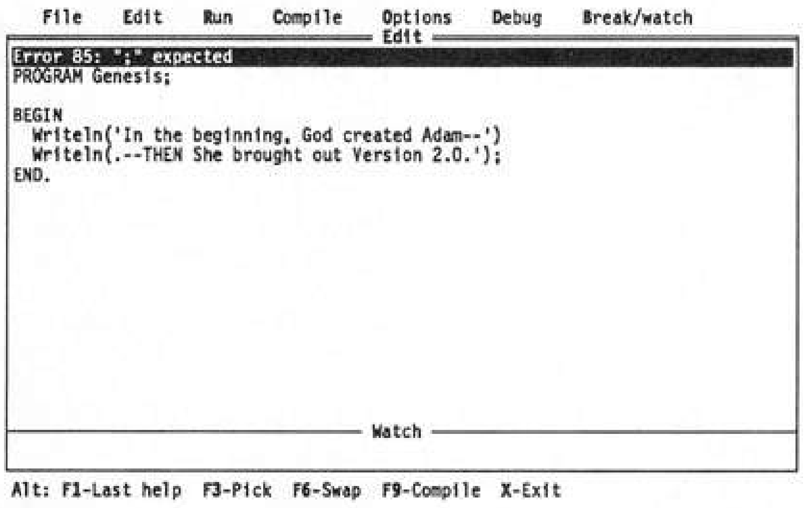
This is perhaps the single most common error ever made by newcomers to Pascal: A semicolon should have been placed at the end of the first **Writeln** statement.

Fixing the error is not difficult. The cursor will be resting under the W in the second **Writeln**. This isn't exactly where the semicolon should go; it is where Turbo Pascal became quite certain that a semicolon was missing. Semicolons serve to separate statements, and there is nothing to separate the two **Writeln** statements. When Turbo ran into the second **Writeln** statement, it realized that no semicolon was going to be encountered, and it started to get confused, hence the error. I'll have a lot more to say about semicolons and keeping them where they belong (and out of where they don't belong) in Section 13.8.

To correct the error, you should move the cursor to the immediate right of the right parenthesis in the first **Writeln** statement and type a semicolon. (The fast way to get there: Press the up arrow once and then press the END key.) Once the semicolon is where it belongs, exit the editor (hint: **CTRL-K/D**), save your corrected program file back out to disk (hint: **F S**), and compile it once again.

Figure 1.4

A Compiler Error Message


File Edit Run Compile Options Debug Break/watch
Edit
Watch
Alt: F1-Last help F3-Pick F6-Swap F9-Compile X-Exit

```
Error 85: ";" expected
PROGRAM Genesis;

BEGIN
  Writeln('In the beginning, God created Adam--')
  Writeln(.--THEN She brought out Version 2.0.');
END.
```

Running

Once a program compiles correctly, it may be run. Under Turbo Pascal 4.0, this is as easy as pressing R. For Turbo Pascal 5.0, you will have to press R twice: Once to get into the **Run** menu, and a second time to select the **Run** command. Program **Genesis** doesn't do much; it only displays these two lines:

```
In the beginning God created Adam--
--and later She brought out Version 2.0.
```

When the program is finished running (which, to your eyes, will be instantaneously), a message will appear at the bottom of the screen reading

```
Press any key to return to Turbo Pascal...
```

It means what it says. Press any key, and the Turbo Pascal Environment will reappear, ready for another cycle of edit, compile, and run.

A Fork in the Road

With that quick overview, you should be able to type in any of the short example programs found in this book, compile them, and run them.

Where to now?

That depends on your frame of mind. The rest of Part One deals with the *idea* of Pascal. There is a philosophy behind the language that goes well beyond the simple stringing of program statements together. If you are a patient, methodical person, and most especially if you want to write large, easily readable programs, I strongly suggest reading the rest of Part One before you do anything else.

If you are fundamentally a practical person and want to learn more about how the system runs, read Part Three now. Part Three is a detailed reference manual for the Turbo Pascal Environment. Much of it is dry reading, but you will have to learn all its material thoroughly to get the most out of Turbo Pascal. *Then*, come back and finish Part One.

Finally, if you are impatient to get down and hack Pascal and are willing to make lots of mistakes in the process, go directly to Part Two and have at it. Part Two is the description of the language itself, feature by feature. Digest that, and the "how" of Pascal will be in your pocket.

I trust you will then return to Part One and find out the "why." By doing so, you will save yourself enormous amounts of time, CRT phosphor, and torn hair. You will also write better programs.

The choice is yours.

2 The Secret Is Structure

By design, Pascal is a structured language. Unlike BASIC and FORTRAN, it imposes a structure on its programs. Pascal will not let you string statements together haphazardly, even if every one, by itself, is syntactically correct. There is a detailed master plan that every Pascal program must follow and the compiler is pretty stiff about enforcing the rules. A program must be coded in certain parts. Some parts must go *here*, and others must go *there*. Everything must be in a certain order. Some things cannot work together. Other things *must* work together.

Aside from some concessions to compiler designers (Pascal makes their task easier in some respects), Pascal's structure exists solely to reinforce a certain way of thinking about programming. This way of thinking represents Niklaus Wirth's emphasis on creating programs that are understandable without scores of pages of flowcharts and thousands of lines of explication. This thought process, championed by Wirth and others, is frequently called "structured programming." Although structured programming can be accomplished in any computer language (even BASIC), Pascal is one of only a few computer languages that *require* it.

2.1: BASIC ELEMENTS OF A PASCAL PROGRAM

A structure must be made of something. A crystal is a structure of atoms in a particular, ordered arrangement. A Pascal program is made of "atoms" that are simply English words formed from the ASCII character set. These program atoms fall into three categories: a very small number of words (only 48) called "reserved words," a much larger number of words called "standard identifiers," and the unlimited multitude of ordinary identifiers created by you, the programmer. Reserved words are words that have special meanings within Pascal. They cannot be used by the programmer except to stand for those particular meanings. The compiler will immediately error-flag any use of a reserved word that is not rigidly in line with that word's meaning. Examples would include **BEGIN**, **END**, **PROCEDURE**, **ARRAY**, and that old devil **GOTO**.

Everything that is not a number or a reserved word is an *identifier*. Some identifiers have predefined meanings to the compiler. These "standard identifiers," like reserved words, have a particular meaning to the compiler. However, under certain circumstances you can redefine their meanings for your own purposes. I shouldn't have to say that redefining a standard identifier is not something you should attempt until you know the Pascal language and your compiler inside and out.

Any name you apply to an entity in a program is an ordinary identifier. The names of variables, of procedures and functions, and of the program itself are identifiers. An identifier you create can mean what you want it to mean (within Pascal's own rules and limits) as long as it is unique. That is, if you have a variable named **Counter**, you cannot have a procedure named **Counter**. Likewise, you cannot have another variable named **Counter**. You can give a particular identifier to only *one* entity of your program. The compiler will flag an error as soon as it spots the second usage. (There is one exception

Table 2.1
Reserved Words in Turbo Pascal 4.0 and 5.0

ABSOLUTE	END	INLINE	PROCEDURE	TYPE
AND	EXTERNAL	INTERFACE	PROGRAM	UNIT
ARRAY	FILE	INTERRUPT	RECORD	UNTIL
BEGIN	FOR	LABEL	REPEAT	USES
CASE	FORWARD	MOD	SET	VAR
CONST	FUNCTION	NIL	SHL	WHILE
DIV	GOTO	NOT	SHR	WITH
DO	IF	OF	STRING	XOR
DOWNTO	IMPLEMENTATION	OR	THEN	
ELSE	IN	PACKED	TO	

to this rule, having to do with a concept called *scope*. More on scope will appear in Chapter 3.)

Combining reserved words and identifiers gives you statements. **IF**, **THEN**, =, and **GOTO** are reserved words. (The equal sign, =, is called an *operator*, which is a special kind of reserved word.) **Flush** is a standard identifier. **Counter**, **Limit**, and **MyFile** are programmer-defined (that is, by you) identifiers.

```
IF Counter = Limit THEN Flush(MyFile);
```

is a statement.

Combining statements in a fashion that respects Pascal's structure gives you a program.

2.2: DATA AND STRUCTURES MADE OF DATA

There are other programming languages that enforce a program structure, such as Algol and PL/1. Pascal goes beyond both in that it allows a programmer to build structures of data as well as structures of program statements. Understanding this will require a look at the Pascal concept of *data*.

Data are chunks of information that your program manipulates. There are different *types* of data, depending on what the data are intended to represent. How your program handles your variables depends completely on what type you decide they are. Every variable used in a Pascal program must be declared to be of some type, with a notation like this:

```
CreditHours : Integer;
```

This will allow you to manipulate integer values in a variable called **CreditHours**, subject to Pascal's explicit limitations on what can be done with integers.

At the lowest level, all data in a Pascal program (or any program, really) are stored as binary numbers somewhere in your computer's memory. The data type dictates to some extent the way those binary numbers are arranged in memory and to a greater extent how you as a human being will use the data. The concept of data type is much more important in Pascal than it is in BASIC. Numbers and strings are the only common data types used in BASIC. Pascal goes much further in the different types of data it recognizes. It defines certain fundamental types of data:

Integers are numbers (including negative numbers) that cannot have decimal points, like 1, −17, and 4529.

Characters (indicated by predefined identifier **Char**) consist of the ASCII character codes from 0 to 127. These include all the common letters, numbers, and symbols used by all modern computers. Turbo Pascal extends that to include those 128 ASCII characters plus an additional 128 foreign language and special symbols.

Boolean type variables have only two possible values: **True** and **False**. They are sometimes called "flags." Pascal uses them in conditional statements like **IF/THEN/ELSE** and **REPEAT/UNTIL** to determine whether to take some action or not.

Real type variables are used to express real numbers, numbers that may include a decimal part. For example, 1.16, −3240, 6.338 and −74.0457 are all real numbers.

In addition to these elementary data types understood by all Pascals, Turbo Pascal adds several of its own. A detailed discussion of all data types appears in Part Two.

Virtually any task can be accomplished using these fundamental data types alone. However, creating structures of data by using these fundamental data types as building blocks can help you develop a program design, and help you code the program once the design is complete. Using the basic data types, Pascal allows a programmer to build special-purpose types that are valid only within the program in which they are defined.

One way to build new special-purpose data types is by defining *subranges*. A subrange is a type that may have as its values only certain values from the range of the fundamental data type. For example, academic grades are usually expressed as letters. Not all letters are grades, however. You might define a subrange of the basic type **Char** that can have as values only the letters from A through F. Such a type would be defined this way:

```
Grade = A..F;
```

Now, to create a variable to hold grades, you would declare a variable this way:

```
History : Grade;
```

Subranges provide a modicum of protection against certain coding mistakes. For example, if you tried to assign a grade of W to the variable **History**, the compiler would tell you during compilation that W is not a legal value of the subrange Grade.

Another powerful tool for building types is called a *record*. A record is a grouping of fundamental types into a larger structure that is given a name as a new type. Variables can be declared to be of this type. Those variables can be assigned, compared, and written to files just as integers or characters can.

For example, if you are writing a program that keeps track of student grades and test results, you might group together basic data types in this arrangement:

```
TYPE
  SemesterGrades = RECORD
                     StudentID   : String[9];
                     SemesterID  : String[6];
                     Math        : Grade;
                     English     : Grade;
                     Drafting    : Grade;
                     History     : Grade;
                     Spanish     : Grade;
                     Gym         : Grade;
                     SemesterGPA : Real
                   END;
```

Now you can define a variable as having the type **SemesterGrades**. By a single variable name, you now can control nine separate data chunks that you otherwise would have had to deal with separately. This can make certain programming tasks a *great* deal simpler. But, more importantly, it allows you to treat logically connected data as a single unit to clarify your program's design and foster clear thinking about its function. For example, when you need to write the semester's grades out to a disk file, you needn't fuss with individual subjects separately. The whole record goes out to disk at once, as though it were a single variable and without any reference to the individual variables from which the record is built. The alternative is a series of statements that write the student ID to disk, followed by the semester ID, followed by the math grade, followed by the English grade, and so forth.

When you need to think of all a student's grades taken together, you can think of them as a unit. When you need to deal with them separately, Pascal has a simple way of picking out any individual variable within the record. (We'll be treating all these subjects in detail in Part Two.) How you think of the data now depends on how you *need* to think of the data. Pascal encourages you to structure your data in ways like this that encourage clear thinking about your problem at a high level (all grades taken together) or at a low level (each grade a separate data item.)

Much of the skill of programming in Pascal is learning how to structure your data so that details are hidden by the structure until they are needed. It is much like being able to step back and see your data as a forest without being distracted by the individual trees.

The structuring of data is a two-edged sword. It is all too easy to create data structures of Byzantine complexity that add nothing to a program's usefulness while obscuring its ultimate purpose. If the data structure you create for your program makes the program *harder* to understand from "three steps back," you've either done it the wrong way, or done it too much.

The rule of thumb I use is this: *Don't create data structures for data structure's sake.* Unless there's a reason for it, resist. Simplicity, when scrutinized, is not a simple thing.

2.3: THE MASTER PLAN OF A PASCAL PROGRAM

The preceding overview described the elements of a Pascal program. The program itself must be constructed from these elements in a very particular way. To some extent, this is done to make life easier for the compiler program. However, the bottom-line reason for the plan of a Pascal program is to make it easy to think of the program's task in a clear, structured way.

A pictorial diagram of the master plan is given in Figure 2.1. You might want to follow along on it as you go. From the top:

The Program Statement

The *program statement* gives your program a name and tells the compiler that the work begins *here.* This name need not be the same as that of your source file nor as that of the machine code file the compiler produces as an end product. This is a program statement:

```
PROGRAM Logbook;
```

The Unit Usage Declaration Part

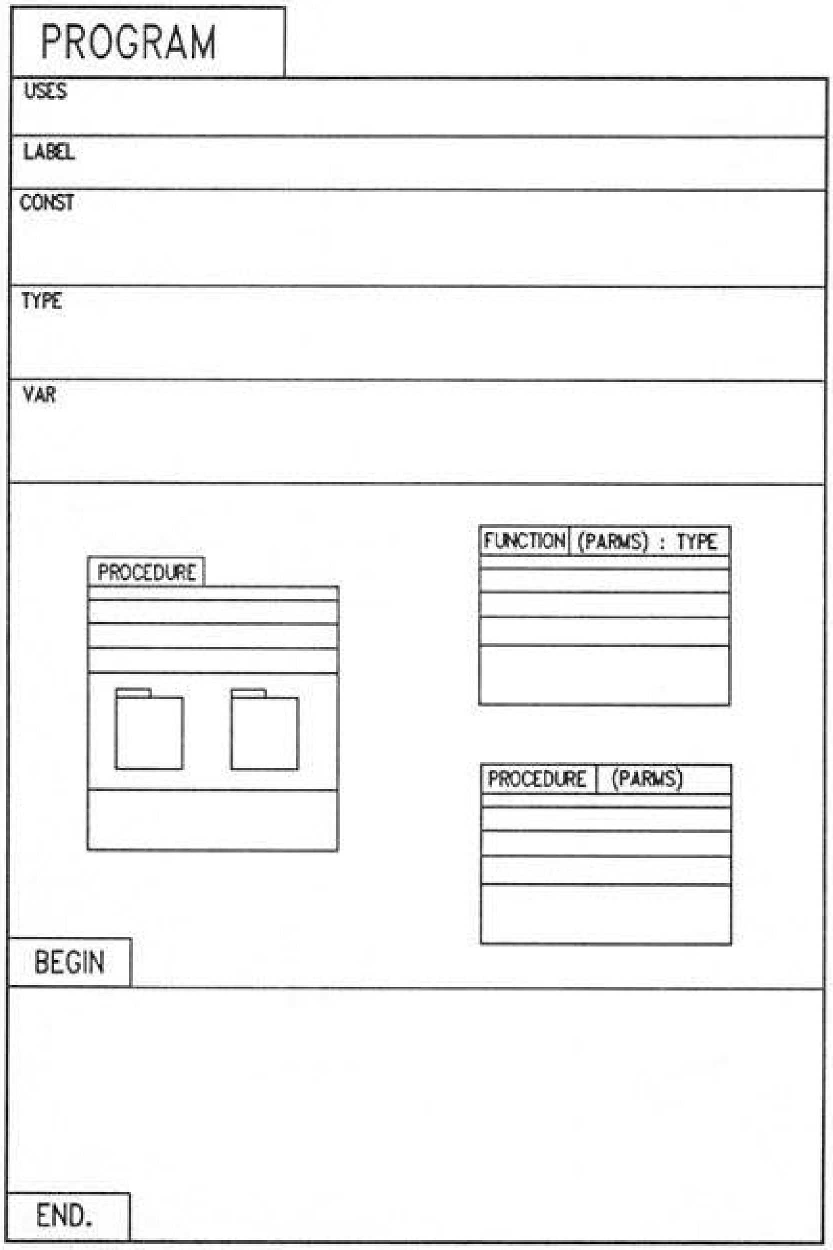
Turbo Pascal differs from ISO standard Pascal in that it allows "separate compilation." (Much more on this appears later, in Chapter 17.) If you wish to make use of a separately compiled module in your program, you must have a *units usage declaration part* in your program. It reads sensibly:

```
USES
  DOS,CRT;
```

meaning, almost literally, that program **LogBook** (assuming the **PROGRAM** statement mentioned above) uses two separately compiled *units* called **DOS** and **CRT**. There are many routines in **DOS** and **CRT** that will save you the trouble of writing them yourself. Later, you can keep your own useful routines in units you write yourself. Most of your

Figure 2.1

The Structure of a Turbo Pascal Program



programs will have a **USES** statement. It is optional, but if it exists, it must be the first noncomment following the **PROGRAM** statement.

The Label Declaration Part

The *label declaration part* of the program comes next. It consists of the reserved word **LABEL** followed by your labels. Labels are markers in your program to which a **GOTO** statement may go. They are whole, non-negative numbers. They need not be in any particular order nor in any particular range. Each one must be unique, however. This is a sample label declaration. (The program it came from probably uses too many GOTOs!)

```
LABEL
  50,100,150,200,250;
```

If you have no **GOTO**s in your program (and that is generally a very good idea), you won't need labels. Not having labels means the label declaration part of your program is unnecessary. But if you have one, it must be the first noncomment following the **USES** statement. If you have no **USES** statement, it must be the first noncomment following the **PROGRAM** statement.

The Constant Declaration Part

If you intend to use any constants in your program, the compiler expects them to follow the label declaration part. Constants are values that are defined at compile time and never change while the program is running. The constant declaration part consists of the reserved word **CONST** followed by your constant declarations. Declaring a constant consists of the constant's identifier and its value, separated by an =.

```
CONST
  Limit      = 255;
  GName      = '3-D Bar Graph';
  UsePlotter = True;
```

In the above example, **Limit** is an integer constant, **GName** is a string constant, and **UsePlotter** is a Boolean constant. Standard Pascal and Turbo Pascal differ significantly in what may be a constant and what may not. We will be speaking of constants in detail in Part Two.

The Type Declaration Part

Following the constant declaration part of your program comes *type declaration.* The type declaration part consists of the reserved word **TYPE** followed by your type declarations. A type declaration is the type identifier and its declaration, separated by an =. Pascal predeclares fundamental types like **Integer**, **LongInt**, **Boolean**, **Char**, **Byte**, and **Real**. You don't have to declare these. You need only declare types that you build from other types, or declare as *subranges* of other types.

You can build types out of other types that you have already defined. However, you can't use a type as an ingredient in a new type unless the compiler already "knows" what that type is. The compiler knows a type either because that type is predeclared (**Integer**, **Byte**, etc.) or because it already has compiled a statement defining that type in terms of other previously known types.

```
TYPE
  TeamTag    = 'A'..'G';               { Subrange }

  Callsign   = String[8];              { "short" string type }

  USHam      = RECORD
                 OpCall    : Callsign;
                 LicClass  : Char;
               END;

  Contact    = RECORD
                 HisCall   : Callsign;
                 RST       : Integer;
                 Band      : Integer;
                 Frequency : Real;
                 Emission  : String[2]
                 Date      : Integer;
                 ZuluTime  : Integer;
               END;

  LogEntry   = RECORD
                 OurOp     : USHam;
                 Team      : TeamTag;
                 QSO       : Contact
               END;
```

Above are a series of type declarations as they might appear in a ham radio logging program. If the details of notation are still strange to you, don't worry. We'll be covering all this material in detail in Part Two. The important thing to notice is that all types are defined in terms of fundamental types or subranges (through a statement containing a =) before they are used.

For example, all the types that make up **LogEntry** are programmer-defined types. Each must have been defined prior to the definition of **LogEntry**, and each was. If **TeamTag** had been defined *after* **LogEntry**, the compiler would have flagged it as an error during compilation of the program. It would not have known what the identifier **TeamTag** stood for. **TeamTag** is actually a subrange of type **Char** (it includes the characters from A to G), and, once it is defined in those terms, the compiler is perfectly comfortable dealing with it.

As with label declarations, the type declaration part is optional. If you have defined no types yourself, you don't need to include any type declarations.

Ordinarily, the constant declaration part of your program comes before the type declaration part. Turbo Pascal allows you to declare constants *after* types. This is done in connection with array constants and record constants (Section 9.6).

The Variable Declaration Part

Following your type declarations must be your variable declarations. Strictly speaking, a program does not need a variable declaration part to compile or run. However, although you might be able to get away without having any label declarations or type declarations, it would not be much of a program without some variables to work with!

The variable declaration part consists of the reserved word **VAR** followed by your variable declarations. Every variable you use must have a valid type, either a predefined type (**Integer**, **Char**, **Boolean**, etc.) or a type you define yourself in the type declaration part of the program. Unlike type declarations, variable declarations use a colon to separate the variable name from its type:

```
VAR
  Counter : Integer;
  Tag     : Char;
  Grid    : XY_Pair;
```

If several variables have the same type, you can group them on one line, separating them by commas like this:

```
VAR
  I,J,K : Integer;
```

We should point out that the reserved word **VAR** may appear many times within a Turbo Pascal program, which is *not* the case with ISO Standard Pascal. However, good practice suggests using the reserved word **VAR** only once.

Procedure and Function Definitions

After all your variables have been declared, you must define any procedures and functions your program is to include. (That is, apart from procedures and functions in a unit that your program uses through the **USES** statement.) Procedures and functions are Pascal's subroutines. In form they are miniature programs, identical in structure to the program itself except that procedures and functions begin with **PROCEDURE** or **FUNCTION** rather than **PROGRAM** and do not end with a period but with a semicolon. Functions, additionally, return a value.

The rest of the structure remains the same. Procedures and functions may have their own private label declarations, constants, programmer-defined types, variables, and even their own procedures and functions. They may not have a **USES** statement, however. This nesting of structures can theoretically go on forever, but there are always implementation restrictions, which means, ultimately, that the machine eventually runs out of RAM to work with. Of course, there are practical, hardware-dictated limits to any compiler feature. Be aware of them as you work out solutions to your programming problems.

The Pascal compiler enforces a *calling hierarchy* among procedures and functions. A procedure or function may call any procedure or function *above* itself in the source file. No procedure or function may call any procedure or function *below* itself without special authorization. (Such authorization is called a "forward declaration," and we'll discuss it later.) The reason is similar to the reason you must define a type before you use it: The compiler is travelling down the source file, and it is building a list of the identifiers it encounters as it proceeds. It will not allow one procedure to call another unless the name of the called procedure is already on that list. This can only happen if the called procedure was encountered earlier by the compiler; in other words, it exists *above* the calling procedure in the source file.

In this hierarchy, the first procedure the compiler encounters can't call anything, and the last procedure can call any procedure or function in the program. It's easy to understand why: all the other procedures and functions are above the last procedure, and the compiler already has all the other procedures on its list by the time it arrives at that last procedure.

Think of it as a hostess travelling down a line of guests, taking names. By the time she meets the last man in line, she already knows the names of all the others and can introduce the last man to anyone else he wishes to meet. But if the poor chap she greets first wishes to know who else is in line, the hostess can only shrug her shoulders. She simply doesn't know yet.

The Main Program

Of course, the main program, being at the very bottom of the source file, is Boss—it calls anybody it wants. As we said above, the main body of your program is treated very much like the body of a procedure or function by the compiler, except that it must always come last, and it must end with a period.

When the computer begins executing your compiled program, it enters the program, not at the top of the source file, but at the **BEGIN** that begins the main program. When it encounters the **END** at the end of the main program, it leaves the program and returns control to DOS. In between, it does all the work the program must do, by calling functions and procedures and following the program's flow of control.

```
PROGRAM Tiny;

CONST
  Ten = 10;

VAR
  I    : Integer;

BEGIN
  I:=Ten;
  Writeln('>>I am now ',I);
END.
```

```
PROCEDURE Tiny;

CONST
  Ten = 10;

VAR
  I    : Integer;

BEGIN
  I:=Ten;
  Writeln('>>I am now',I);
END;
```

There's not much difference between **PROGRAM Tiny** and **PROCEDURE Tiny**, is there? Aside from the reserved words **PROGRAM** and **PROCEDURE** and the period or semicolon at the end, there is none. The two useless little things have the same structure and do the same work. The *real* difference between them is that only **PROGRAM Tiny** can stand on its own. **PROCEDURE Tiny** *must* exist within some other program, or the compiler will kick it out as an error.

The structure of a procedure or function is actually the structure of a program in miniature. A Pascal program is much like a collection of nested Chinese boxes, one (or more) inside another, all cubical and made of the same material. But only one box is big enough to contain all the others, and that box is always on the outside. You can only get at the inside boxes by picking up the largest box and opening it.

The following program is a real, functioning program. It performs a nonsense function on a text file, producing a "foo factor" for any given text file. It's not intended to be useful, nor is the programming method ideal. (No program this small should have **LABEL**s or **GOTO**s. I put them in to show you how such creatures fit into program structure.) Ignore what it does for now, and simply look at how the different parts are put together.

Look first at the main program. It's short—only a few statements long. The statements are relatively general ones, mostly function and procedure calls. Reading down the main program, you can quickly understand what the program has to do: Open a file, calculate a foo factor for that file, display the foo factor, and close the file. Notice that the details for opening the file or calculating the foo factor are not given in the main program. They don't have to be there. In fact, if they were there it would be harder to understand what the program does.

To find those details, you have to go up into the list of procedures and functions. Function **CrunchFile** calculates the foo factor for the entire file. It does this by reading lines from the file and calculating a foo factor for each line. But calculating a foo factor for a line is a relatively involved process in itself. Those details are therefore extracted and put into yet another function called **CrunchLine**. By putting the details for crunching a line into a separate procedure, we make the process for crunching the whole file easier to understand.

CrunchFile can call **CrunchLine**, and does. But **CrunchLine** cannot call **CrunchFile**. **CrunchLine** lies above **CrunchFile** in the source file and in the hierarchy. **CrunchLine** could call **OpenAFile** if it needed to, but it does not.

```
PROGRAM Generic;


LABEL
  100;

CONST
  Iterations = 25;

TYPE
  InputFile  = TEXT;
  String80   = String[80];

VAR
  Counter    : Integer;
  OK         : Boolean;
  LIMIT      : Integer;
  FooFactor  : Integer;
  Buffer     : String80;
  DoFile     : InputFile;


PROCEDURE OpenAFile(VAR OK : Boolean);

VAR
  I : Integer;

BEGIN
  Assign(DoFile,'MYTEXT.TXT');
  {$I-} Reset(DoFile); {$I+}
  I := IOResult;
  IF I = 0 THEN OK := True ELSE OK := False;
END;


FUNCTION CrunchLine(Buffer : String80) : Integer;

VAR
  I      : Integer;
  Bucket : Integer;

BEGIN
  CrunchLine := 0;
  IF Length(Buffer) > 0 THEN
```

```
    BEGIN
      Bucket := 0;
      FOR I := 1 TO Length(Buffer) DO Bucket := Bucket +
        Ord(Buffer[I]);
      Bucket := Bucket DIV LENGTH(Buffer);
      CrunchLine := Bucket
    END
END;


FUNCTION CrunchFile(VAR DoFile : InputFile) : Integer;

VAR
  Passes : Integer;
  Temp   : Integer;

BEGIN
  Temp := 0;
  FOR Passes := 1 TO Iterations DO
    BEGIN
      Reset(DoFile);
      WHILE NOT EOF(DoFile) DO
        BEGIN
          Readln(DoFile,Buffer);
          Temp := Temp+CrunchLine(Buffer);
        END
    END;
  CrunchFile := Temp DIV Iterations
END;



BEGIN    { Main Program }
  OpenAFile(OK);
  IF NOT OK THEN
    BEGIN
      Writeln('>>The file cannot be opened.');
      GOTO 100
    END;
  FooFactor := CrunchFile(DoFile);
  Writeln('>>Foo Factor for Input File is ',FooFactor);
  Close(DoFile);
  Writeln('>>Processing completed.');
  100:
END.
```

The calling hierarchy forces the details of the structure to the top and the broad strokes of the program structure to the bottom. As you move upward from the main program, the level of detail increases. This gives you a clear method to begin understanding a Pascal program: Read and understand the main program first. This gives you the broadest picture of the program with the fewest details to distract you. From there, move upward to the major procedures and functions. Move upward even more, and by the time you understand the topmost procedures and functions on the source file, you will have it all, and you will have learned it in the right order, the order most conducive to clear thought about the program and what it does.

This sort of approach to designing and understanding programs—large structure first, details last—has often been called "top-down programming." Pascal, to a large extent, enforces a top-down method on its users. My objection to the term top-down programming lies in that, with respect to a Pascal source file, it is really "bottom-up programming." Your design and understanding really works upward from the bottom of the source file.

I prefer to think of a Pascal program as a pyramid. You must climb it (and *build* it!) from the base upward. If you attempt to build the details at the top first without a well-designed base to rest them on, the whole thing will collapse all over you.

This, in brief, is the master plan of a Pascal program. Some programmers object to the admittedly rigid nature of this plan. I'll be the first to admit that it makes clever hacks difficult to achieve. On the other hand, if you feel, as I do, that it is important to produce code that is as easy to understand (and fix) next year as it is today, you'll design your programs along the Pascal plan, even without a whip-wielding compiler to enforce it.

Remember: The pyramids are not an especially clever hack. But they've been around a *long* time.

3

Nesting and Scope

The notion of *scope* in a Pascal program is often a peculiar and slippery one to beginning programmers, and it's worth a little space all by itself.

Scope is an attribute of an identifier, be it variable, constant, data type, procedure, or function. The scope of an identifier is the region of a program that "knows" that identifier and can gain access to it. It has to do with levels of nesting.

As we saw in the previous section, a program can be thought of as a Chinese box containing smaller programs built to the same general Pascal structure. These smaller programs are the main program's procedures and functions. They are nested within the main program.

What we didn't emphasize (for clarity's sake) was that any procedure or function can have other procedures and functions nested within it. Like the Chinese boxes, this nesting of procedures and functions can continue until your computer system runs out of memory to contain it all.

The constants, variables, procedures, functions, or other identifiers that are declared within a procedure or function are said to be "local" to that procedure or function.

```
PROGRAM Hollow;

VAR
  Z     : Integer;
  Ch, Q : Char;
  Gonk  : String[80];


PROCEDURE LITTLE1;

VAR
  Z : Integer;

BEGIN
END;


PROCEDURE LITTLE2;

VAR
  Z : Integer;
  Q : Char;

BEGIN
END;



BEGIN     { Main for Hollow }
  Little1;
  Writeln('>>We are the hollow programs,');
  Little2;
  Writeln('>>We are the stuffed programs.')
END.
```

The program **Hollow** is nothing more than the merest skeleton of a program, constructed to illustrate the concept of scope and nesting. If you're sharp and have looked closely at **Hollow**, you may be objecting to the fact that there are three instances of a variable named **Z**, and we have already seen that Pascal does not tolerate duplicate identifiers. Well, because of Pascal's notion of scope, the three **Z**s are not duplicates at all.

Up near the top of the source file, in **Hollow**'s own variable list, is a variable named **Z**. In **Little1** is a variable named **Z**, but *this* **Z** is local to **Little1**. **Little2** also has a **Z** that is local to **Little2**. Each **Z** is known only in its own neighborhood, and the extent of that neighborhood is its scope. The scope of **Little1**'s **Z** is *only* **Little1**. Likewise, **Little2**'s **Z** is known only within **Little2**. You *cannot* access the value of **Little2**'s **Z** (or, for that matter, *any* variable declared within **Little2**) from within **Little1**. Furthermore, while you're in the main program, you cannot access either of the two **Z**s that are local to **Little1** and **Little2**. They might as well not exist until you enter one of the two procedures in which those local **Z**s are declared.

This becomes a little slipperier when you consider the **Z** belonging to program **Hollow** itself. That **Z**'s neighborhood encompasses the entire program, which includes both **Little1** and **Little2**. So, while we're within **Little1** or **Little2**, which **Z** is the real **Z**? Plainly, we need a rule here, and the rule is called "precedence." *When the scopes of two identical identifiers overlap, the most local identifier takes precedence.* In other words, while you're within **Little1**, **Little1**'s own local **Z** is the only **Z** you can "see." The **Z** belonging to **Hollow** is hidden from you while you're within the scope of a more local **Z**. **Hollow**'s **Z** doesn't go away and doesn't change; you simply can't look at it or change it while **Little1**'s **Z** takes precedence.

Try this metaphor. There are too many Mike Smiths in the world. A national liquor company based in New York City has a vice president named Mike Smith. The company also has two regional salesmen named Mike Smith, one in Chicago and one in Geneseo, New York. When you're at corporate headquarters in New York and mention Mike Smith, everyone assumes you mean the Vice President of Whiskey Keg Procurement. However, if you're in Chicago and mention Mike Smith to a liquor store owner, he thinks you mean the skinny chap who sells him Rasputin Vodka. He's never heard of the vice president or the salesman in Geneseo. Furthermore, if you're in Geneseo and mention Mike Smith, the restaurant owners think of the fellow with the red beard who distributes Old Tank Car wines. They don't know, and couldn't care less about, the company vice president or the vodka salesman in Chicago.

Look back at **Hollow** for a moment and consider the variable **Q**. **Hollow** has a **Q**. **Little2** also has one. **Little1** does not. Within **Little2**, **Little2**'s **Q** is king, and **Hollow**'s **Q** is hidden away. However, **Little1** can read and change **Hollow**'s **Q**. There is no precedence conflict here because there is no **Q** in **Little1**. Since the scope of **Hollow**'s

Q is the entire program, any function or procedure within **Hollow** can access **Hollow**'s **Q** as long as there is no conflict of precedence. We say that **Hollow**'s **Q** is *global* to the entire program. In the absence of precedence conflicts, **Hollow**'s **Q** is known throughout **Hollow**.

Hollow has a variable named **Gonk** that can be accessed from either **Little1** or **Little2** because it is the only **Gonk** anywhere within **Hollow**. With **Gonk**, the question of precedence does not arise at all.

Unless you can't possibly avoid it, don't make your procedures and functions read or change global variables (e.g., **Ch** or **Gonk**) *unless* those variables are passed to the procedure or function through the parameter line. This prevents the creation of data "sneak paths" among your main program and procedures and functions. Such sneak paths are easy to forget when you modify a program, and may bollix up legitimate changes to the program in apparently inexplicable ways.

To sum up: An identifier is local to the block (procedure, function, module, or program) in which it is defined. This block is the *scope* of that identifier.

You cannot access a local identifier unless you are *within* that block or within a block that *contains* that block.

Where a duplicate identifier conflict of scope exists, the *more local of the two* is the identifier that you can access.

4 Programming by Understanding the Problem First

4.1: THE PROBLEM OF IMPATIENCE

The number one killer of computer programs is impatience. The number one waster of data processing dollars is impatience. The number one mass-producer of program bugs is impatience. If you can conquer impatience, you have computer programming, in Pascal or any other language, right in your hip pocket.

The problem is simple enough. Planning is dull and difficult. Coding is fun. Because of a natural human tendency, planning often gets short shrift during a programming project. Programmers dive into coding long before they have a really clear notion of what the project entails. The project definition, or "requirements," may not even be finished before the coding begins.

The hazards of working this way should be obvious, but a real-world (and true) example should make them abundantly clear:

A school district set out to create a student database. The programmers, working with a handwritten data structure sketched out in half an hour by the project manager, began work, with the excuse that keying in the data would take weeks. They decided that having a "little module" to accept data would allow the keypunchers to get going. The little module was done in a week, and the keypunchers start punching. Days and thousands of student entries later, the project manager realized that he would need several new fields on each student to comply with Federal student aid regulations. The incomplete data structure was, at that point, enshrined in 2000 lines of code and 4000 records of data. The addition of the support for the additional data fields was done hastily, and a second "little module" was created to accept the keypunchers' input for the missing fields. The patches to the code proved bug-ridden, and the matching the of additional data to the student records was faulty. The bottom line (when all the tangles are untangled) was three months of additional and unnecessary effort to get the system running. Bugs are still haunting the users 3 years after implementation.

Knowledgeable people in the software engineering field claim that there is a certain irreduceable amount of effort required to design and implement a computer system. If that effort is not expended in planning, it will be expended two, three, or many times over in scrapping and rewriting program modules and patching what the writers do not have the intestinal fortitude to scrap and rewrite.

Sadly, many textbooks on programming start new programmers off on the wrong foot, by showing them on page 2 this single line program:

```
10 PRINT "HI! I'M A BASIC PROGRAM!"
```

On page 3, they add another line of text output. On page 4, they add some sort of numerical calculation. On page 5, they add a branch of some sort. So it goes, through all the features of BASIC, tacking line after line onto the poor program. The purpose is to reinforce the tutorial on BASIC by showing the programmer how the statements work. The lesson that isn't supposed to come across, but does, is that it's perfectly all right to design and write a program by sticking little pieces together without any planning beforehand. Not one word (or maybe a paragraph or two on page 279) is given to the process of planning a program before the coding begins. It doesn't surprise me that impatience ruins so many programming efforts. Impatience is built into the way we teach our programmers their craft.

There is a method of programming championed by Niklaus Wirth that works extremely well on small to medium-sized programming projects. It's called "programming by successive refinement." The gist of the method is this: Begin with a single clear statement of the purpose of the program you want to write, and then gradually refine that statement (breaking it down into subsidiary statements) until the last statement is compilable code in your language of choice. The first several iterations can, and should, be written without your assuming you know the language in which the program will eventually be written.

Because this method begins with a single broad statement and adds detail to the statement in a structured, orderly fashion, it fits in particularly well with the Pascal language.

I have found that the method breaks down when the system is large enough to encompass several very large programs, many necessary manual operations, and concurrent processes. If you find yourself faced with creating such a system, you are much better off following the method of structured analysis and design as outlined by Yourdon and Constantine[1]. This is *especially* true if your system is to be written and implemented by a team of programmers in addition to yourself.

For working on your own in developing reasonably straightforward programs, though, successive refinement will do nicely. The example I'm about to show you is a real, useable program written in Turbo Pascal. If you're reading this book in a strictly serial fashion without any previous knowledge of Pascal, you may not fully understand all the code used. Don't worry about it right now. Look past the actual code and pay closer attention to the *method* involved in developing the code. Then come back after you've read Part Two, which explains the Pascal language in detail, and read this short section again.

4.2: AN EXAMPLE OF SUCCESSIVE REFINEMENT

Even the longest programming project begins with a single step. It's a big one, and an important one: *Formulate a one-sentence statement of what the program must do.*

Make it *one* sentence. Why? It will help you decide what belongs in the program and what does not. In the early, muddled stages of formulation, it often is unclear just

[1]Edward Yourdon and Larry L. Constantine, *Structured Design* (Englewood Cliffs, NJ: Prentice-Hall, Inc., 1979).

what a program has to do. Should all the work you desire be done by a single program? Do the various tasks belong together?

If you cannot describe the program in broad terms with fewer than two sentences, see if the tasks represented by the two sentences have anything in common. If not, you're better off splitting your single proposed program into two or checking to see if you're beginning to describe how the process is *to be done*, rather than stating what the process *is*. In that case, you have not used the broadest terms to describe your program. Look to your sentences and work backward from the *how* to the *what*.

Now, back to our example. To keep the example program down to manageable size, its task has to be relatively simple: *Chart the frequency of different word lengths appearing in a text file.*

Program Statement Iteration 1:

Collect and display the relative frequency of different word lengths appearing in a text file.

This is the first iteration of your program statement, the single-sentence statement emphasized above. Very simply, you want to see which lengths of words appear most frequently in various textfiles you happen to have.

This will involve reading the text file, tallying the instances of each word length, and, finally, displaying the tallies in a meaningful way. In fact, the previous sentence lies at the heart of iteration 2. You might have been tempted to use this sentence as your single-sentence program statement, but you should not. Yes, it's a single sentence, but it tells *how* to do something, not simply what needs to be done. Rule of thumb: Once you begin talking about *how* instead of *what*, you're already beyond the first iteration.

Program Statement Iteration 2:

1. Select a text file and open it.
2. Set up a counter for each reasonable word length.
3. Scan the text file line by line, incrementing the appropriate counter for the length of each word read from the file.
4. Display histograms for each word length, reflecting the relative frequency of word lengths in the file.

At each stage in the process, look hard at each step in your current iteration and ask yourself, "Do I really know what each statement means?" If you don't understand the broad sense of each of your steps, you can hardly be expected to break each step down into the more detailed descriptions that your next iteration will demand.

When you reach iteration 3 (perhaps a little later if the problem is large and complex), your program statement will begin to take on some of the characteristics of your chosen language. Look for statements that can become procedures and functions. Begin to express the necessary data as real data items, even if they are not completely, rigorously defined in their first appearance. Indicate loops with indentation.

Avoid the temptation to write code at this point. You're not quite ready yet.

Program Statement Iteration 3:

```
Opener(FileName, ErrorFlag)
  Open the file.
  If an error occurs, return an error flag.

Grapher(Counters)
  For each word length:
    Display the length as a number.
    Display a line of * s proportional to counter


MAIN PROGRAM
  Get a filename from somewhere.
  Opener(FileName, ErrorFlag)
  If ErrorFlag, print error message and exit.
  Set up an array of 50 integers.
  Until you hit EOF on the file:
    Read a line.
      Until the line is empty:
        Identify the first word in the line.
        Determine the length of the word.
        Increment that length's counter in array.
        Delete the word from the line.
  Close the file.
  Grapher(Counters)
  Display a message indicating process is complete.
```

Iteration 3 is still not anything like code, although it's plain that we're heading in the direction of Pascal. Keep refining the statement, watching for hidden "gotchas" that indicate some incomplete area of your understanding of the problem.

For example, there's a "gotcha" in iteration 3 that you'll find when you begin to refine the statement for the **Grapher** procedure. **Grapher** must display a line of asterisks proportional to the value in each of your counters. If you scan a large file, you may find that the counter for four-letter words (the polite kind, of course) has a value of several hundred. Your printer prints only eighty columns. You have to *scale* the counters, so that the line of asterisks for the largest counter will still fit across your printer's page.

You must decide how this scaling function is going to be done and where it must fit into the program. You may find, in working out the details, that you have defined a few more program steps and a few more data items. Add these to your next iteration. For example, a file and its name are two separate data items, yet iteration 3 treats them as though they were one. In your next iteration you must deal with both the "logical" (Pascal) and physical (disk resident) file.

In the process you'll discover that here and there things have begun to fall into real Pascal code. One of the first places this will happen is with loops. Once you understand how a loop is to be tested and terminated, it's natural to write that loop's statement as though it were Pascal code. Again, if you are not truly comfortable committing a statement to real code, it may mean you are unsure of what that statement means and how it works.

Program Statement Iteration 4:

```
Variables
  I,J          : general purpose integers
  Counters     : array of 40 integers
  FileName     : a string
  TestFile     : a text file
  Line         : a string
  AWord        : a string
  WordLength   : an integer
  Scale        : a real number

KillJunk(AWord)
  BEGIN
    Take off bad characters in front of the word
    Take off bad characters at the end of the word
  END

Opener(FileName, ErrorFlag)
  BEGIN
    Associate TestFile with FileName
    Open TestFile for read.
    If an error occurs, return an error flag.
  END

Grapher(Counters, Scale)
  BEGIN
    FOR I = 1 TO 40 DO
      BEGIN
        Display word length I
        Display counter / Scale asterisks
      END
  END
```

```
FUNCTION Scaler(Counter) returns real
  BEGIN
    Scan CounterS for largest counter
    IF largest count < printer width THEN exit
      ELSE Scaler := largest count / printer width
  END

BEGIN MAIN PROGRAM
  Extract FileName from the DOS command tail
  Opener(FileName, TestFile, ErrorFlag)
  IF ErrorFlag is true, THEN
    BEGIN
      print error message
      exit
    END
  Set up an array of 40 integers.
  WHILE NOT EOF on TestFile
    BEGIN
      Read Line
      Increment LineCount
        WHILE there's still words in Line DO
          BEGIN
            Extract AWord from Line
            KillJunk(AWord)
            Increment the total word counter
            Limit length of humungous words to 40
            Increment counter for AWord's length
            Delete AWord from Line
          END
    END
  Close TestFile
  Scale := Scaler(Counters)
  Grapher(Counters, Scale)
  Display a message indicating process is complete.
END
```

Iteration 4 is starting to look something like a structured program. The first edition of this book and a great many Pascal programs you'll find in other books and on user group disks show identifiers as well as reserved words in upper case. There's nothing sacred about upper or lower case in Pascal. The compiler forces all program text into upper case before it attempts to compile it. You may write a program completely in upper case, lower case, or mixed case, as you like. The programs in this book reflect the mixed-case conventions that prevail in Turbo Pascal circles. Reserved words remain in upper case, and all identifiers, including predefined identifiers, are in mixed case. Many people "spread out" their identifiers by spacing them with underscores:

```
Error_Flag
Return_To_DOS
```

This is legal, but I do not do it because I also program in Modula 2, which does not allow underscores in identifiers.

The scaling problem was dealt with by adding a function named **Scaler** that returns a scale factor in the variable named **Scale**. The version of **Scaler** in iteration 4 has a logic bug (not a code bug—this isn't code yet) in it. Can you find it? If you can't, compare that expression of **Scaler** with the final Pascal code below.

An additional procedure has been added to take care of removing punctuation, tab characters, and other "junk" from the file, where "junk" is defined as anything that is not a countable letter or number in a word. The exclamation point after "Goodness!" should not be counted as part of the word, nor should the quotes around the word "protection" in "copy 'protection.'"

Deciding when an iteration can take the final leap into code depends heavily on how fully you understand how every part of the program must work and on how comfortable you are with Pascal itself. Above all else, avoid this trap: You are still a little fuzzy on how some detail in the program is supposed to work, so you make a fuzzy guess as to how it should be coded, making a mental note that "this probably won't work, but I'll pound it into shape after the rest of it works." *That's impatience talking!* A fuzzily understood part of your program will invariably propagate its fuzziness to other parts of the program that depend upon it. Bugs that are the consequence of fuzzy design are the hardest to find and the most difficult to remove.

The more confident you are in your knowledge of Pascal, the larger the jump may be from your last iteration to actual code. If you are unsure about Pascal, it's best to take it in smaller jumps. Don't let impatience to get on with the job inspire false confidence.

When you decide you are are ready, pull out your Pascal reference and copy your penultimate iteration to the final iteration (either on paper or on disk), while making sure that all statements have been converted to compilable Pascal code. Print a hard copy of the program, and you can finally turn the compiler loose on it.

The final program in Pascal is shown below. I am only printing five iterations, both for brevity's sake and because it took me five iterations to produce the program. A beginner might move a little more slowly and do the job in seven or eight iterations. Certainly, one additional iteration beyond 4 would be a good idea if you are unsure of Pascal's syntax.

Again, if you don't fully understand all the details of the code in the final program, don't worry about it just yet. All actual code details will be covered in Part Two.

```

{---------------------------------------------------------------}
{                            WordStat                           }
{                                                               }
{      Word Counter & Word Length Tabulator for TextFiles       }
{                                                               }
{                                by Jeff Duntemann              }
{                                and Hugh Kenner                }
{                                Turbo Pascal V5.0              }
{                                Last update 7/14/88            }
{                                                               }
{                                                               }
{                                                               }
{---------------------------------------------------------------}

PROGRAM WordStat;

USES Printer;

CONST
  PrintWidth  = 68;
  Tab         = #9;


TYPE
  Array40     = ARRAY[0..40] OF Integer;
  String80    = String[80];

VAR
  I,J         : Integer;
  Scale       : Real;
  Ch          : Char;
  Opened      : Boolean;
  TestFile    : Text;
  FName       : String80;
  Counters    : Array40;
  Line        : String80;
  AWord       : String80;
  WordLength  : Integer;
  LineCount   : Integer;
  WhiteSpace  : SET OF Char;
  GoodChars   : SET OF Char;


PROCEDURE KillJunk(VAR AString : String80);

BEGIN
  WhiteSpace := [#8,#9,#10,#12,#13,#32];
  GoodChars  := ['A'..'Z','a'..'z','0'..'9'];
  REPEAT        { Clean up leading end of word }
    IF Length(AString) > 0 THEN
      IF (AString[1] IN WhiteSpace) OR (NOT(AString[1] IN GoodChars))
        THEN Delete(AString,1,1)
  UNTIL ((NOT (AString[1] IN WhiteSpace)) AND (AString[1] IN GoodChars))
    OR (Length(AString) <= 0);
  REPEAT        { Clean up trailing end of word }
    IF Length(AString) > 0 THEN
      IF (AString[Length(AString)] IN WhiteSpace)
        OR (NOT(AString[Length(AString)] IN GoodChars))
      THEN Delete(AString,Length(AString),1)
  UNTIL ((NOT(AString[Length(AString)] IN WhiteSpace)
    AND (AString[Length(AString)] IN GoodChars))
    OR  (Length(AString) <= 0))
END;  { KillJunk }



PROCEDURE Opener(    FileName : String80;
                 VAR TFile    : Text;
                 VAR OpenFlag : Boolean);

VAR
  I : Integer;

BEGIN
  Assign(TFile,FileName);       { Associate logical to physical }
  {$I-} Reset(TFile); {$I+}     { Open file for read    }
  I := IOResult;                { I <> 0 = File Not Found  }
  IF I = 0 THEN OpenFlag := True ELSE OpenFlag := False;
END;  { Opener }



FUNCTION Scaler(Counters : Array40) : Real;

VAR
  I,MaxCount : Integer;

BEGIN
  MaxCount := 0;           { Set initial count to 0 }
  FOR I := 1 TO 40 DO
    IF Counters[I] > MaxCount THEN MaxCount := Counters[I];
  IF MaxCount > PrintWidth THEN Scaler := PrintWidth / MaxCount
    ELSE Scaler := 1.0;    { Scale=1 if max < printer width}
END;  { Scaler }



PROCEDURE Grapher(Counters : Array40; Scale : Real);

VAR
  I,J : Integer;

BEGIN
  FOR I := 1 TO 40 DO
    BEGIN
      Write(Lst,'[',I:3,']: ');      { Show count }
      FOR J:=1 TO Round(Counters[I] * Scale) DO Write(Lst,'*');
      Writeln(Lst,'')                { Add (CR) at end of *'s}
    END
END;


BEGIN   { WordStat Main }

  FName := ParamStr(1);             { We must pick up command tail first, }
  KillJunk(FName);                  {   before opening any files! }
  FOR I:=0 TO 40 DO Counters[I]:=0;           { Init Counters }
  LineCount := 0;

  Opener(FName,TestFile,Opened);  { Attempt to open input file  }
  IF NOT Opened THEN                { If we can't open it...      }
    BEGIN
      Writeln('>>>Input file ',FName,' is missing or damaged.');
      Writeln('   Please Check this file''s status and try again.');
    END
  ELSE                              { If you've got a file, run with it! }
```

```
    BEGIN
      WHILE NOT EOF(TestFile) DO  { While there's stuff in the file }
        BEGIN
          Readln(TestFile,Line);        { Read a Line }
          LineCount := LineCount + 1;   { Count the Line }
          Write('.');                   { Display a progress indicator }
          FOR I := 1 TO Length(Line) DO
            IF Line[I] = Tab THEN Line[I] := ' ';
          WHILE Length(Line) > 0 DO     { While there are words in the Line }
            BEGIN
              KillJunk(Line);           { Remove any non-text characters }
              IF POS(' ',Line) > 0 THEN
                AWord := Copy(Line,1,POS(' ',Line)) ELSE AWord := Line;
              KillJunk(AWord);          { Clean up the individual word }
              Counters[0] := Succ(Counters[0]);     { Count the word }
              WordLength := Length(AWord);
              IF WordLength > 40 THEN WordLength := 40;
              J := Counters[WordLength]; { Get counter for that Length }
              J := Succ(J);              { Increment it...     }
              Counters[WordLength] := J; { ...and put it back. }
              Delete(Line,1,Length(AWord));  { Remove the word from the Line }
            END
        END;
      Writeln;
      Close(TestFile);                { Close the input file }
      { The count itself is done.  Now to display it: }
      Scale := Scaler(Counters);      { Scale the Counters }
      Writeln(Lst,
      '>>Text file ',FName,
      ' has ',Counters[0],
      ' words in ',LineCount,' lines.');
      Writeln(Lst,
      '  Word size histogram follows:');
      Grapher(Counters,Scale);        { Display Scaled histograms  }
      Writeln(Lst,Chr(12));           { Send a formfeed to printer }
    END
END.
```

Part Two

Learning the Language

INTRODUCTION

In Part One of this book, we took a roller-coaster ride through the idea of Pascal with the idea of getting a bird's eye view of the larger issues of the language and without stopping too long for individual details.

Now, it's time for the details.

At this point, I want to emphasize the difference between a language *definition* and a language *implementation*. In 1971, Niklaus Wirth wrote the language *definition* of Pascal, in a well-known (and nearly impossible to read) book entitled *Pascal: User Manual and Report*.[1] The book contains a meticulously detailed description of what Pascal should do. However, at that time there was no extant compiler program that would actually take a Pascal source file and produce a runnable code file from it. Someone (many someones, actually) had to take Wirth's book and write a compiler program that obeyed the language definition as Wirth set it out. Such programs first appeared for large mainframe computers, such as the CDC 6600, and have appeared for microcomputers only in the last 10 years or so.

Those compiler programs that functionally embody the Pascal language definition are called "implementations" of the Pascal language.

Wirth's formal definition of the Pascal language has a lot of holes and weak spots that limit its use. Wirth's definition does not address string handling, random files, operating system calls, and many other things that we take for granted today. People who actually implement the language on a particular computer usually expand beyond the language definition to make the compiler program capable of compiling more useful programs.

Furthermore, some implementations are forced to limit the programmer in ways that Wirth's definition does not. To make use of small memory systems, real-world limits must be placed on identifier size, procedure and function size, maximum legal values for integers and real numbers, maximum member counts for sets, and so on. None of these things are present in the definition of the language, but all are critical to their particular implementations.

Thus, while Part One dealt mostly with the language's definition, Part Two describes a very particular implementation: Turbo Pascal 4.0 and 5.0. To allow you to do useful programming with this implementation of Pascal, we have to deal with implementation details, such as data range limitations, language extensions, and many other things.

It's important to remember, especially if you find yourself using other compilers on other computers, that Pascal as implemented in Turbo Pascal is only broadly the same as Pascal as implemented in the UCSD P-System, MS Pascal, or OMSI Pascal 2. The details are likely to be different.

And details, in computer science, are everything.

[1]Niklaus Wirth and Kathleen Jensen, *Pascal: User Manual and Report* (New York: Springer Verlag, 1974).

5 Identifiers and the Naming of Names

God created the animals, and Adam gave them names.

The animals were, after all, for Adam's use, and he had to have some way of keeping them all straight. God doesn't have problems like that.

Your computer creates programs for your use. Programs are collections of things (data) and steps to be taken in storing, changing, and displaying those things (statements). The computer recognizes such things by their addresses in memory. The readable, English-language names of data, programs, and functions and procedures are for your benefit. We call such names, taken as a group, *identifiers.* They exist in only the source code. *Identifiers do not exist in the final code file.*

Identifiers are sequences of characters of any length up to 127 characters that obey these few rules:

1. Legal characters include letters, digits, and underscores. Spaces and symbols, such as &, !, *, or % (or any other symbol that is not a letter or digit), are not allowed.
2. A symbol or a digit (0 through 9) may *not* be used as the first character in an identifier. All identifiers must begin with a letter from A through Z (uppercase or lowercase) or with an underscore.
3. Identifiers may not be identical to any reserved word.
4. Case differences are ignored. "A" is the same as "a" to the compiler.
5. Underscores are legal and significant. Note that this differs from most other Pascal compilers, in which underscores are legal but ignored. You may use underscores to spread out a long identifier and make it more readable: **SORT_ON_ZIP_CODE** rather than **SORTONZIPCODE**. (A better method, which has become the custom in the Turbo Pascal community, is to use mixed case to accomplish the same thing: **SortOnZIPCode**.)
6. *All* characters are significant, no matter how long the identifier is, up to 127 characters. Many other Pascals allow but ignore any character after the eighth character in an identifier.

These are all invalid identifiers that will generate errors:

```
Fox&Hound       Contains an invalid character
FOO BAR         Contains a space
7Eleven         Begins with a number
Do@Noon         @ is valid only at the beginning!
RECORD          "RECORD" is a reserved word
```

6 Simple Constants

Constants are data values that are "baked into" your source code and do not change during the execution of a program. There are two kinds of simple constants in Standard Pascal: literal and named. Turbo Pascal provides a third kind of constant that is not really a constant: typed constants, that is, constants that have a specified type and can be data structures like arrays and records. We will discuss typed constants in Section 9.6.

6.1: LITERAL VERSUS NAMED CONSTANTS

A *literal constant* is a value of some sort that is stated as a value where it is used in your code. For example:

```
SphereVolume := (4/3)*PI*(Radius*Radius*Radius);
```

In this line of code, "4" and "3" are literal constants, representing the values of 4 and 3.

There is another constant in that statement: The identifier **PI** was previously declared a constant in the constant declaration part of the program:

```
CONST
  PI = 3.14159;
```

PI is a *named constant.* We could as well have used the literal constant 3.14159 in the statement, but **PI** is shorter and makes the expression less cluttered and more readable. Especially where numbers are concerned, named constants almost always make a program more readable.

Another use for constants is in the setting of program parameters that need changing only very rarely. They still *might* be changed someday. If you use a named constant, changing the constant *anywhere in the program* is only a matter of changing the constant's declaration *once* in the constant declaration part of the program and then recompiling.

The alternative is to hunt through hundreds or thousands of lines of source code to find every instance of a literal constant to change it. You will almost certainly miss at least one, and the resultant bug farm may cost you dearly.

In short, don't use literal constants anywhere you anticipate *ever* needing changes. In mathematical formulae, literal constants are usually OK; the value of **PI** hasn't changed recently, certain state legislatures notwithstanding.

6.2: CONSTANTS AND THEIR TYPES

In Standard Pascal, constants may be simple types and sets only. These include real numbers, integers, bytes, characters, strings, sets, and Booleans. Individual enumerated types may also be considered constants, although they are not declared in the same way other constants are. (We'll speak more fully of enumerated types in Section 7.6.) Structured types such as records, pointers and arrays may *not* be constants in Standard Pascal. Turbo Pascal supports data structure constants (see Section 9.6), but we must cover data structures first before we can speak of them.

Here are some sample named constants of various types:

```
CONST
  PI           = 3.14159;       { Floating point real  }
  Threshold    = -71.47;        { Negative FP real     }
  PenIOAddress = $06;           { Hexadecimal value    }
  Using8087    = True;          { Boolean              }
  DriveUnit    = 'A';           { Character            }
  Revision     = 'V2.61B';      { String               }
  Answer       = 42;            { Integer              }
  NotAnswer    = -42;           { Negative integer     }
  YesSet       = ['Y','y'];     { Set                  }
  NullString   = '';            { Null (empty) string  }
  BigNum       = 6117834        { Long integer or real }
```

Constants Versus Data

How is a constant different from a variable? The obvious difference is that the value of a constant is set at compile time. You cannot assign a value to a constant. Given the list of constants above, you could not legally code:

```
Answer := 47;
```

because **Answer** is a constant with a value of 42.

In Turbo Pascal, simple constants are written into the code by the compiler as immediate data. Variables are kept separate from the code portion of a program. Constants, therefore, do not take up room in your data segment.

The type of a constant depends, to some extent, on its context. Consider:

```
VAR
   Tiny   : Byte;         { One byte                   }
   Little : Integer;      { An integer (2 bytes)       }
   Big    : LongInt       { A long integer (4 bytes)   }
   Huge   : Real;         { A real number (6 bytes)    }


   Tiny   := Answer;
   Little := Answer;
   Big    := Answer;
   Huge   := Answer;
```

From the constants above, **Answer**'s value is 42. It is perfectly legal, though, to assign the value of **Answer** to type **Byte**, type **Integer**, type **LongInt**, or type **Real**. In each case, the code the compiler generates to do the assignment is a bit different, but the end result is that all three variables of three different types will each express a value of 42 in its own fashion.

Except under very special (and peculiar) circumstances, the type of a variable is fixed and unambiguous.

Notes on Literal Constants

A dollar sign ($) in front of a numeric literal means that the compiler will interpret the literal as a hexadecimal number. The numeric literal may *not* have a decimal point if it is to be considered hexadecimal.

Inside string literals, lowercase and uppercase characters are distinct. If you wish to include a single quote mark inside a string literal, you must use two single quotes together:

```
Writeln('>>You haven''t gotten it right yet...');
```

This line of code will display the following line on your CRT:

```
>>You haven't gotten it right yet...
```

6.3: CONSTANT EXPRESSIONS (VERSION 5.0)

With release 5.0, Turbo Pascal added the concept of *constant expressions*. Prior to release 5.0, a named constant could be defined in only one way: by stating its name and equating it to a single literal value:

```
THR = $3F8;
```

A constant expression allows you to give a value to a named constant in terms of an expression that is evaluated at compile time. I haven't covered "expressions" yet, but you may look ahead to Chapter 11 if you would like to understand constant expressions fully at this point; otherwise, return to this section once you've had a chance to use and understand expressions.

Note that only simple constants may be given values from constant expressions. *Typed constants may not be assigned values from constant expressions.*

An expression, briefly, is a combination of identifiers, values, and operators that "cooks down" to a single value. Expressions resemble portions of equations from physics into which you plug necessary values and from which you finally evaluate a single value:

Kinetic energy = Mass * Velocity^2

Once you know **Mass** and **Velocity**, you can plug their values into the equation and do the math to come up with a value for **Kinetic energy**.

It's much the same way with constant expressions. The value of a constant is given in terms of a combination of operators, values, and constants defined earlier in the program:

```
COMPORT  = 1;        { 1 = COM1:  2 = COM2: }
COMBASE  = $2F8;     { Build on this "magic number" }

{ Base I/O port is $3F8 for COM1: and $2F8 for COM2: }
PORTBASE = COMBASE OR (COMPORT SHL 8);
```

Here, **COMPORT** and **COMBASE** are simple named constants, whereas the constant **PORTBASE** is defined in terms of a constant expression. When the compiler encounters the constant expression during its compilation of the program, it does the math and assigns the resulting value to the named constant **PORTBASE**.

Limitations of Constant Expressions

If you're familiar with expressions as they occur in normal Pascal statements, you may be wondering if *any* legal expression may be assigned to a constant. The answer is emphatically *no;* constant expressions are much more limited. Legal elements of a constant expression are these:

- *Literal constants.* These include numeric literals like 42 and 17.00576, the Boolean literals **True** and **False**, and quoted string and character literals.
- *Previously defined constants.* In other words, named constants defined earlier in the program, like **COMPORT** and **THRBASE** in the example above.
- *Pascal operators.* These are the arithmetic operators like addition and subtraction, the logical operators like AND and OR, and the bitwise operators like AND, OR, SHR, SHL, and so on. For a complete discussion of Turbo Pascal's operators, see Section 11.
- *Certain built-in functions.* A very few of Turbo Pascal's built-in functions may take part in constant expressions. These include **Ord**, **Chr**, **Odd**, **Hi**, **Lo**, **Length**, **Abs**, **Pred**, **Succ**, and **Swap**. All other functions, including those you write and those built into the compiler, such as **Sqr** and **Cos**, are illegal. For those of you who can appreciate the differences, Turbo Pascal generates code in-line for "functions" like **Abs**, while other functions like **Sqrt** and **Cos** are true functions that must be called from the runtime library. **Abs** and its kin are more properly *macros* than functions and can be evaluated in-line by the compiler during compilation.

Although it might be obvious to veteran Pascal hackers, it's worth stating clearly that *variables cannot be part of a constant expression.* Turbo Pascal allows constants to be defined after variables are declared (as Standard Pascal does not) but not after variables are assigned values. This means that, at best, a variable would introduce an undefined quantity into a constant expression, which is far less than useful.

Furthermore, as mentioned above, typed constants cannot be assigned values from constant expressions.

Some Examples of Constant Expressions

The best way to show what's possible with constant expressions is to put a few in front of you. The following are all legal, if not necessarily useful in every case.

```
CONST
  Platter   = 1;
  FirstSide = Odd(Platter);   { Boolean }
  FlipSide  = NOT FirstSide;  { Boolean}

  Yesses    = ['Y','y'];      { Character set }
  Noes      = ['N','n'];      { Character set }
  Answers   = Yesses + Noes;  { Intersection of 2 sets }

  USHoller  = 'ATTENTION!!  ';      { String }
  USGasMsg  = 'Fuel level is low!'; { String }
  USGasWarn = USHoller + USGasMsg;  { String concatenation }
  USWarnSiz = Length(USGasWarn);    { String length }

  LongSide  = 17;
  ShortSide =  6;
  TankDepth =  8;
  Volume    = LongSide * ShortSide * TankDepth;
```

Why Use Constant Expressions?

There are two excellent reasons to use constant expressions: reconfiguration and documentation. Both relate to the use of what I call "magic numbers" in program development.

Many programs use values that may be defined once in a program and are never modified. These include mathematical constants as well as I/O port numbers and bit numbers for low-level control of system hardware. These magic numbers aren't expected to change, but, to be safe, they should always be defined as constants rather than written as dozens or hundreds of literals "shotgunned" throughout a program.

An excellent example of reconfiguration through constant expressions involves programming the communications port on the PC. The PC supports two serial ports, COM1: and COM2:. They are accessed through different sets of I/O ports, and the differences in the port addresses follow an unchanging relationship. By defining a single constant specifying either COM1: or COM2:, the control port addresses may be recalculated in constant expressions based on that single port number definition. This is what the following code (part of which was given earlier) does:

```
CONST
  COMPORT  = 1;        { 1 = COM1:  2 = COM2: }
  COMBASE  = $2F8;  { Build on this "magic number" }

  { Base I/O port is $3F8 for COM1: and $2F8 for COM2: }
  PORTBASE = COMBASE OR (COMPORT SHL 8);

  { Transmit Holding Register is write-only at the base port: }
  THR = PORTBASE;

  { Receive Buffer Register is read-only at the base port: }
  RBR = PORTBASE;

  IER = PORTBASE + 1;  { Interrupt Enable register }
  IIR = PORTBASE + 2;  { Interrupt identification register }
  LCR = PORTBASE + 3;  { Line control register }
  MCR = PORTBASE + 4;  { Modem control register }
  LSR = PORTBASE + 5;  { Line status register }
  MSR = PORTBASE + 6;  { Modem status register }
  SCR = PORTBASE + 7;  { Scratch register }
```

Every one of the constants defined in this code fragment has a different value, depending on whether the COM1: or COM2: serial port is to be used. By changing the value of the constant **COMPORT** from 1 to 2, *all* the other constants change accordingly to the values that apply to serial port COM2:. The program does not need to be peppered with magic numbers like $2FC and $3FA. Also, your program does not need to spend time initializing all these port numbers as variables, because the compiler does all the calculation at compile time. The resulting values are inserted as immediate data into the code generated from your source file.

The other use for constant expressions is in helping your programs document themselves. You may need some sort of mathematical "fudge factor" in a complicated program. You can define it as a simple named real-number constant:

```
Fudge = 8.8059;
```

No one looking at this value would have any idea of its derivation. If the value is, in fact, the result of an established formula, it can help readability to make the formula part of a constant expression:

```
ZincOxideDensity = 5.606;
Fudge = ZincOxideDensity * (PI / 2);
```

This will help others (or even you) keep in mind that you had to fudge things by multiplying the density of zinc oxide by Pi over 2. (That is, assuming you want the world to know. . . .)

The idea should never be far from your mind that Pascal programs are meant to be read. If you can't read them, you can't change them or fix them. You might as well throw them away and start from scratch. Do whatever you can to make your programs readable. You (or your eventual replacement) will be glad you did.

7 Ordinal Data Types

Data are the pieces of information that your program manipulates. Data can be numbers, characters, character strings, and other easy-to-visualize symbols. They also can be highly abstract, like conditions of Boolean truth or falsehood, sets of concepts like days of the week, or elaborate structures, called *records*, built out of simpler items.

The richness of expression with which Pascal can treat data places it far apart from earlier languages like BASIC and FORTRAN. Modern versions of BASIC may have strings and different types of numerics, but few BASICs allows you to build complex data structures from simple data types.

The *type* of a data item is actually a set of rules governing the storage and use of that data item. Data take up space in RAM memory. The type of a data item dictates how much space is needed and how the data item is represented in that space. An integer, for example, always occupies 2 bytes of memory. The most significant bit of an integer carries the sign of the number the integer represents.

Type also governs how a data item may be used. Type **Char** and type **Byte** are both single bytes in RAM, but you can add or multiply two variables of type **Byte**. Attempting to use variables of type **Char** in an arithmetic expression will generate an error.

The Pascal language uses *strong typing*, which means that there are strict limitations on how individual types may be used, especially on how variables of one type may be assigned to variables of another type. In most cases, a variable of one type may not be assigned to a variable of a different type. Transferring information between variables across type boundaries is usually done with transfer functions that depend on well-defined relationships between types. Transfer functions will be described in Chapter 16.

Simple types (which embrace all types described in this section) are "unstructured." That is, they are data "atoms" that cannot be broken down into simpler data types.

7.1: CHARACTERS

Type **Char** (character) is an ISO Standard Pascal type and is present in all implementations of Pascal.

Type **Char** includes the familiar ASCII character set: letters, numbers, common symbols, and the "unprintable" control characters like carriage return, backspace, tab, and so on. There are 128 characters in the ASCII character set. Type **Char**, though, actually includes 256 different values, because a character is expressed as an 8-bit byte. (Eight bits may encode 256 different values.) The "other" 128 characters have no names or meanings as standard as those of the ASCII character set. When printed to the CRT of the IBM PC, the "high" 128 characters display as foreign language characters, segments of boxes, or mathematical symbols.

How, then, do you represent characters in your program? The key lies in the concept of *ordinality*. There are 256 different characters included in type **Char**. These characters exist in a specific ordered sequence numbered 0,1,2,3, and onward up to

255. The 65th character (counting from 0, remember) is always capital A. The 32rd character is always a space, and so on.

An ordinal number is a number indicating a position in an ordered series. A character's position in the sequence of type **Char** is its ordinality. The ordinality of capital A is 65. The ordinality of capital B is 66, and so on. Any character in type **Char** can be unambiguously expressed by its ordinality, using the ISO standard transfer function **Chr**.

Capital A may be expressed as the character literal A. It may also be expressed as **Chr(65)**. The expression **Chr(65)** may be used anywhere you would use the character literal A.

Beyond the limits of the ASCII character set, the **Chr** function is the only reasonable way to express a character. The character expressed as **Chr(234)** will display on the IBM PC screen as the Greek capital letter omega (Ω) but it will probably be displayed as something else on another computer. It is best to express such characters using the function **Chr**.

Characters are stored in memory as single bytes, expressed as binary numbers from 0 through 255.

What will Pascal allow you to do with variables of type **Char**?

1. You can write them to the screen or printer using **Write** and **Writeln**:

```
Writeln('A');
Write(Chr(234));
Write(UnitChar);    { UnitChar is type Char }
```

2. You can concatenate them with string variables using the string concatenation operator (+) or the **Concat** built-in function (see Section 15.1):

```
ErrorString := Concat('Disk error on drive ',UnitChar);
DriveSpec   := UnitChar + ':' + FileName;
```

3. You can derive the ordinality or characters with the **Ord** transfer function:

```
BounceValue := 31+Ord(UnitChar);
```

   Ord returns a numeric value giving the ordinality of the character parameter. **Ord** allows you to perform arithmetic operations on the ordinality of a character.
4. You can compare characters to one another with relational operators like =, >, <, >=, <=, and <>. This is because of the way characters are ordered in a series. What you actually are comparing is the ordinality of the two characters in their series when you use relational operators. For example, when you see:

```
'a' > 'A'
```

(which evaluates to a boolean value of **True**), the computer is actually performing a comparison of the ordinalities of a and A:

```
97 > 65
```

Because a is positioned *after* A in the series of characters, its ordinality is larger, and therefore a is in fact "greater than" A.

7.2: BOOLEANS

Type **Boolean** is part of ISO Standard Pascal. A Boolean variable has only two possible values, **True** and **False**. Like type **Char**, type **Boolean** is an ordinal type, which means it has a fixed number of possible values that exist in a definite order. In this order, **False** comes before **True**. By using the transfer function **Ord**, you would find that:

Ord(False)	returns the value 0.
Ord(True)	returns the value 1.

The words **True** and **False** are predefined identifiers in Pascal. The compiler predefines them as constants of type **Boolean**. As with any predefined identifier, the compiler will allow you to define them as something other than Boolean constants, but that is a thoroughly bad idea.

A Boolean variable occupies only a single byte in memory. The actual words **True** and **False** are not physically present in a Boolean variable. When a Boolean variable contains the value **True**, it actually contains the binary number 01. When a Boolean variable contains the value **False**, it actually contains the binary number 00. If you write a Boolean variable to a disk file, the binary values 00 or 01 will be physically written to the disk. However, when you print or display a Boolean variable using **Write** or **Writeln**, the binary values are recognized by program code, and the words **TRUE** or **FALSE** (in upper case ASCII characters) will be substituted for the binary 00 and 01.

Boolean variables are used to store the results of expressions using the relational operators =, >, <, <>, >=, and <=, and the set operators +, *, and −. (Operators and expressions will be discussed more fully in Chapter 11.) An expression such as **2 < 3** is easy enough to evaluate. Logically you would say that the statement "two is less than three" is true. If this were put as an expression in Pascal, the expression would return a Boolean value of **True**, which could be assigned to a Boolean variable and saved for later processing:

```
OK := 2 < 3;
```

This assignment statement stores a Boolean value of **True** into the Boolean variable **OK**. The value of **OK** can later be tested with an **IF..THEN..ELSE** statement, with different actions taken by the code depending on the value assigned to **OK**:

```
OK := 2 < 3;
IF OK THEN
  Writeln('>>Two comes before three, not after!')
ELSE
  Writeln('>>We are all in very serious trouble...');
```

Boolean variables are also used to alter the flow of program control in the **WHILE..DO** statement and the **REPEAT..UNTIL** statement (see Sections 13.5 and 13.6).

Defining Your Own Data Types

All the types we have discussed up to this point have been simple types, predefined by Turbo Pascal and ready to use. Much of the power of Pascal lies in its ability to create structures of data out of these simple types. Data structures can make your programs both easier to write and, later on, easier to read.

Defining your own custom data types is easy to do. The reserved word **TYPE** begins the type definition part of your program, and that's where you lay out the plan for your data structures:

```
TYPE
  YourType = ItsDefinition;
```

From now on, many of the types we're going to discuss must be declared and defined in the type definition part of your program.

A type definition, by itself, does not occupy space in your program's data area. A type definition provides instructions to the compiler telling it how to deal with variables of type **YourType** when it encounters them further down in your program source file.

With some exceptions (strings and subranges, for example), you cannot write structured types to your screen or printer with **Write** or **Writeln**. If you want to display structured types somehow, you must write procedures specifically to display some representation of the type on your CRT or printer.

7.3: SUBRANGES

The simplest data structure you can define is a subset of an ordinal type called a *subrange*. If you choose any two legal values in an ordinal type, those two values plus all the values that lie between them define a subrange of that ordinal type. For example, these are subranges of type **Char**:

```
TYPE
  Uppercase = 'A'..'Z';
  Lowercase = 'a'..'z';
  Digits    = '0'..'9';
```

Uppercase is the range of characters A, B, C, D, E, F, and so on to Z. **Digits** includes the numeral characters 0, 1, 2, 3, 4, and so on to 9. The quotes are important. They tell the compiler that the values in the subrange are of type **Char**. If you left the quote marks out of the type definition for type **Digits**:

```
Digits = 0..9;
```

you would have, instead, a subrange of type **Integer**. The character 7 is not the same as the digit 7!

An expression in the form of **'A'..'Z'** or **3..6** is called a *closed interval*. A closed interval is a range of ordinal values that includes the two stated boundary values and all values falling between them. We will return to closed intervals in Chapter 9.

The Turbo Pascal compiler stores subranges in the most compact possible form. For example, type **Integer** occupies 2 bytes, but if an integer subrange has 255 elements or fewer, the compiler will allocate only a single byte to that subrange.

7.4: ENUMERATED TYPES

Newcomers to Pascal frequently find the notion of enumerated types hard to grasp. An *enumerated type* is an ordinal type that you define. It consists of an ordered list of values with unique names. One of the best ways to approach enumerated types is through comparison with type **Boolean**.

Type **Boolean** is, in fact, an enumerated type that is predefined by the compiler and used in special ways. Type **Boolean** is an ordered list of two values with unique names: **False** and **True**. It is *not* a pair of ASCII strings containing the English words "False" and "True." As we mentioned earlier, a Boolean value is actually a binary number with a value of either 00 or 01. We "name" the binary code 00 within type **Boolean** as **False**, and name the binary code 01 within type **Boolean** as **True**.

Consider another list of values with unique names: The colors of the spectrum. (Remember that colorful chap Roy G. Biv?) Let's create an enumerated type in which the list of values includes the colors of the spectrum, in order:

```
TYPE
  Spectrum = (Red, Orange, Yellow, Green, Blue,
              Indigo, Violet);
```

The list of an enumerated type is always given within parentheses. The order in which you place the values within the parentheses defines their ordinal value, which you can test using the **Ord(X)** function. For example, **Ord(Yellow)** would return a value of 2. **Ord(Red)** would return a value of 0.

You can compare values of an enumerated type with other values of that same type. It may be helpful to substitute the ordinal value of enumerated constants for the words that name them when evaluating such comparisons. The statement **Yellow > Red** (think: 2 > 0) would return a Boolean value of **True**. **Green > Violet** and **Blue < Orange** would return Boolean values of **False**.

The values of type **Spectrum** are all constants. They may be assigned to variables of type **Spectrum**. For example:

```
VAR
  Color1, Color2 : Spectrum;

Color1 := Yellow;
Color2 := Indigo;
```

You cannot, however, assign anything to one of the values of type **Spectrum**. **Red := 2** or **Red := Yellow** make no sense.

Enumerated types may index arrays. For example, each color of the rainbow has a different frequency of light associated with it. These frequencies could be stored in an array, indexed by the enumerated type **Spectrum**:

```
Wavelength : ARRAY[Red..Violet] OF Real;
Frequency  : ARRAY[Red..Violet] OF Real;
Color      : Spectrum;
Lightspeed : Real;


Wavelength[Red]    := 6.2E-7;     { All in meters }
Wavelength[Orange] := 5.9E-7;
Wavelength[Yellow] := 5.6E-7;
Wavelength[Green]  := 5.4E-7;
Wavelength[Blue]   := 5.15E-7;
Wavelength[Indigo] := 4.8E-7;
Wavelength[Violet] := 4.5E-7;
```

The functions **Ord** and **Odd** work with enumerated types, as do the **Succ** and **Pred** functions. This is because an enumerated type has a fixed number of elements in a definite order that does not change. **Succ(Green)** will return the value **Blue**. **Pred(Yellow)** returns the value **Orange**. Be aware that **Pred(Red)** and **Succ(Violet)** are undefined. You should test for the two ends of the **Spectrum** type while using **Succ** and **Pred** to avoid assigning an undefined value to a variable.

Enumerated types may also be control variables in **FOR/NEXT** loops. In continuing with the example begun above, we might calculate the frequencies of light for each of the colors of type **Spectrum** this way:

```
Lightspeed := 3.0E08               { Meters/second }
FOR Color := Red TO Violet DO
Frequency[Color] := Lightspeed / Wavelength[Color];
```

One great disadvantage to enumerated types is that they cannot be printed to the console or printer. You cannot, for example, code up:

```
Writeln(Orange);
```

and expect to see the ASCII word "Orange" appear on the screen. Turbo Pascal will flag this as:

```
Error 64: Cannot Read or Write variables of this type
```

In cases in which you must print an enumerated type to the screen or printer, set up an array of strings that is indexed by the enumerated type:

```
Names : ARRAY[Red..Violet] OF String80;

Names[Red] := 'Red';
Writeln(Names[Red]);
```

The **Writeln** statement above will print the string "Red" to the screen. You might also set up an array constant containing the names of the items in an enumerated type. This feature is unique to Turbo Pascal and will make your programs thoroughly nonportable, so caution is advised. Array constants are covered in Section 9.6.

8 Numeric Types

Turbo Pascal 4.0 and 5.0 add considerable richness to the Standard Pascal suite of numeric types. Types **ShortInt**, **Word**, **LongInt**, **Single**, **Double**, **Extended**, and **Comp** are all new and did not exist in version 3.0. With all that power come a few problems and certainly a lot more to remember.

8.1: BYTES

Numeric type **Byte** is *not* present in ISO Standard Pascal, although most microcomputer implementations of Pascal include it. Type **Byte** may be thought of as an unsigned "half-precision" integer. It may express numeric values from 0 to 255. Like **Char**, **Byte** is stored in memory as a byte (8 bits). On the lowest machine level, therefore, **Byte** and **Char** are exactly the same. They only differ in what the compiler will allow you to do with them.

Byte variables may not share an assignment statement with any type other than **Integer**, **Word**, **Shortint**, or **Longint**. Mixing type **Byte** with **Boolean**, **Char**, or any other non-numeric type will be flagged as:

```
Error 26: Type Mismatch
```

However, type **Byte** may be freely included in expressions with the other numeric types described in this section. Type **Byte** may not be assigned, however, a numeric value of type **Real**, **Single**, **Double**, **Extended**, or **Comp**.

Range Errors

Variables of type **Byte** may *not* take on negative values. If you try to assign a negative constant to a variable of type **Byte**, you will get this message:

```
Error 76: Constant out of range
```

Assigning a signed variable (like **Integer**, **ShortInt**, or **LongInt**) to **Byte** is perfectly safe *unless the signed variable contains a negative value.*

The compiler will not detect the problem when you compile your program. Difficulties will appear at run time; that is, when you actually run the program you have written. Depending on whether you have enabled range checking during the compilation of the program, two things may happen:

If range checking was *on*, you will see a run time error:

```
Runtime error 201 at 3101:0058
```

The two numbers at the end are the segment and offset addresses of the point in your object code at which the error occurred and will be different depending on your machine and your program. If you received the error while running your program from within the Turbo Pascal Environment, you can press any key and the Environment will send you back into the Editor. At the top of the screen you will see the message

```
Error 201: Range check error
```

and the cursor will be flashing at the point in the code at which the error occurred. Of course, if you were simply executing a stand-alone program from DOS, the more cryptic error message will be all you'll have to go on.

If range checking was *not* on when you compiled the program, the program, in essence, will "punt." It will do its best to pour a signed value into an unsigned variable, and what ends up in the variable depends on the physical bit-pattern of the negative value. To put it mildly, such errors are unpredictable, and, because they happen in statements that often work perfectly well, they can take a *great* deal of time to locate and fix.

This kind of error, known generically as a "range error," will also happen if you attempt to assign a value larger than 255 to type **Byte**. In general, each numeric type has a defined range. If you enable range checking from the **O**ptions menu, assigning a value outside that range to a variable will generate a runtime error. What we've just said applies to the other numeric types discussed below as well as to type **Byte**.

8.2: SHORT INTEGERS

First cousin to type **Byte** is type **ShortInt**, a signed version of **Byte**. It may express values between −128 and 127. The problems of range errors exist for **ShortInt** just as they do for type Byte. It exists to provide a little bit of efficiency to programs that use a lot of small, signed values. If you use a lot of numeric variables or (especially) large arrays of numeric variables and can be sure the values will never wander out of the range −128 through 127, you can save a lot of space by using **ShortInt** variables instead of type **Integer**.

Bit 7 of a **ShortInt** is the sign bit (Figure 8-1). If this bit is set to 1, the value is considered to be negative.

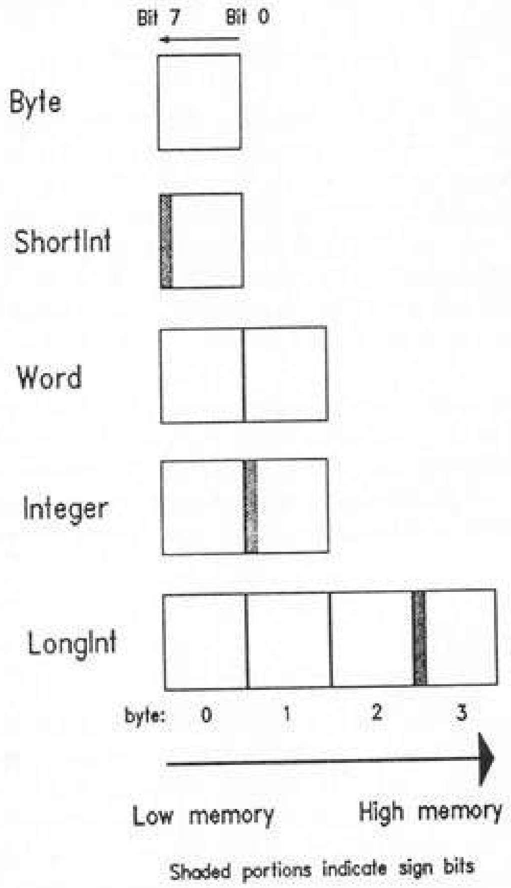
I have not detected any significant speed improvement by using **ShortInt** over other numeric types, however.

8.3: INTEGERS

Type **Integer** is a part of ISO Standard Pascal. Integers may express a range of values from −32768 to 32767. Integers are always whole numbers. They cannot have decimal parts. In Pascal, only real number types (**Real**, **Single**, **Double**, **Extended**, and **Comp**) can have decimal parts.

Figure 8.1

Memory Representation of Integer Types



Integers are stored in memory as two bytes (see Figure 8.1). The highest-order bit of the two higher bytes in memory is the sign bit, which indicates whether the value expressed by the integer is positive or negative. If this high bit is a binary 1, then the integer is negative. If the high bit is a binary 0, the integer is positive.

Hi and Lo

You separate an integer into its 2 bytes by using a pair of built-in functions: **Hi** and **Lo**. As you might expect, **Hi** returns the higher of the integer's 2 bytes in memory, and **Lo** returns the lower. For example, given the integer 17,353 (hexadecimal equivalent $43C9):

```
Hi(17353) will return 67 (hex $43)

Lo(17353) will return 201 (hex $C9)
```

Note that in using **Hi** and **Lo** on integers, the sign bit is treated as just another bit in the high byte returned by **Hi** and will not cause the value returned by either **Hi** or **Lo** to be returned as a negative quantity. For example, given the negative integer constant −21,244 (hex $AD04):

```
Hi(-21244) will return 173 (hex $AD)
Lo(-21244) will return 4 (hex $04)
```

MaxInt

There is a predefined identifier **MaxInt**, which is a constant containing the maximum value an integer may express: 32767.

8.4: WORDS

Turbo Pascal added an unsigned partner to the **Integer** type with Version 4.0. This is type **Word**, and it expresses positive values from 0 through 65535. Niklaus Wirth added a similar type to his Modula 2 language but called it **Cardinal** instead of **Word**.

Like type **Integer**, type **Word** exists in memory as 2 bytes. The low-order portion of the word is contained in the lower of the two in memory, as shown in Figure 8.1. Also, like type **Integer**, you can separately access the 2 bytes by using the **Hi** and **Lo** functions:

```
VAR
  HiByte,LoByte : Byte;
  MyWord        : Word;


HiByte := Hi(MyWord);
LoByte := Lo(MyWord);
```

8.5: LONG INTEGERS

In Turbo Pascal 3.0, the largest number of objects you could count was only 32,767. That might seem like a lot, but when you consider the computer's own ability to handle millions of bytes of RAM memory and tens or even hundreds of millions of bytes of disk storage, it suddenly looks like small change indeed. Turbo Pascal added the **LongInt** type to Version 4.0 to deal with the pressing need to count many things.

Type **LongInt** has the range of −2,147,483,648 through 2,147,483,647. This can handle any memory or disk system we're likely to be able to afford for another few months.

LongInt is implemented as 4 bytes (see Figure 8.1) with bit 7 of the highest order byte used as the sign bit.

MaxLongInt

Turbo Pascal 4.0 adds a predefined constant of type **LongInt** named **MaxLongInt**. This has the value 2,147,483,647.

8.6: FLOATING POINT REAL NUMBERS

All the data types described up to this point have been ordinal types. Ordinal types are types with a limited number of possible values, existing in a definite order. Integers are of the ordinal type, because there are exactly 65,535 of them. They are ordered and sharply defined: After 6 comes 7, after 7 comes 8, and so on, with no possible values in between. Integers have absolute precision; that is, the value of the integer 6 is exactly six.

The real world demands a way to deal with fractions. To do this, ISO Standard Pascal supports type **Real**, which can express numbers with fractions and exponents. Real numbers, especially very large ones or very small ones, do not have absolute precision. For example, **1.6125E10** is a real number having an exponent. You might expand the exponent and write this number as 16,125,000,000. This notation implies that we know the value precisely, but we do not. A real number offers a fixed number of significant figures and an exponent giving us an order of magnitude, but there is a certain amount of "fuzz" in the value. The digits after the 5 in 16,125,000,000 are zeroes because we do not know what they really are. The measurements that produced the number were not precise enough to pin down the last six digits, so they were left as zeroes to express the order of magnitude expressed in the exponential form **1.6125E10**.

Real numbers are "real" in that they usually are used (in the scientific and engineering community) to represent measurements made of things in the real world. Integers, by contrast, are largely mathematical in nature and express abstract values usually generated by logic and not by physical measurement.

Real numbers may be expressed two ways in Turbo Pascal. One way, as we have seen, is with a mantissa (e.g., 1.6125) giving the significant figures and an exponent (e.g., E10) giving the order of magnitude. This form is used for very large and very small numbers. For very small numbers, the exponent would be negative: **1.6125E−10**.

You would read this number as "one point six one two five times ten to the negative tenth."

The second way to express a real number is with a decimal point: **121.402**, **3.14159**, **0.0056**, **−16.6**, and so on.

Math Coprocessor Floating Point Types

Type **Real** is represented in memory as 6 bytes, giving real numbers a range of 10^{-38} to 10^{38} with eleven significant figures. Type **Real** is always available in Turbo Pascal 4.0, but if your system includes a math coprocessor like the 8087, 80287, or 80387, a number of additional real number types become available for use. These types depend on the presence of the math coprocessor, and Verson 4.0 code that uses them will not run correctly (or possibly not at all!) on machines without a math coprocessor installed. (Version 5.0 fixes this problem and allows you to use the math coprocessor types whether you have a math coprocessor or not.)

Whether or not the compiler will accept statements that use one of these coprocessor-dependent floating point types depends on two things:

- The **$N** Numerics compiler directive must be active. Because the default for **$N** is passive, you *must* place the **{$N+}** directive at the beginning of your source file to avoid the following error:

```
Error 116: Must be in 8087 mode to compile this
```

- The compiler must detect a math coprocessor in the system it is running on. Note that variables of one of the coprocessor-dependent types may be *defined* regardless of the state of the **$N** directive or the presence or absence of a coprocessor; errors occur *only* if one of the defined variables actually is used within a program or subprogram **BEGIN/END** block.

  The coprocessor-based floating point types are fully compatible with the IEEE floating point specification, which is a well-established industry standard way of expressing floating point values in memory.

The IEEE floating point types differ considerably in size and range:

- **Single** is a "single-precision" real number type. It is implemented in four bytes, and has a range of 10^{-38} to 10^{38}. This is the same *range* as the 6-byte *Real* type, but because **Single** is implemented as only 4 bytes instead of 6, it has less *precision* and will only yield six or seven significant figures. Calculations incorporating **Single** values will occur significantly faster than equivalent calculations using type **Real**, because of both **Single's** smaller size and its implementation in the silicon of the coprocessor.
- **Double** is a "double-precision" real number type. This type is identical to the 8-byte real number format used in the now-obsolete Version 3.0 Turbo-87 Pascal for the

PC. *If you need to read real numbers written to a file using Turbo-87 Pascal, read them into variables of type **Double**.* Type **Double** has a range of 10^{-307} to 10^{307}, with 16 significant figures of accuracy.

- **Extended** implements what the 8087 refers to as a *temporary real.* It is expressed as 10 bytes in memory and has the astonishing range of 10^{-4932} to 10^{4932} with 19 significant figures of accuracy. This is the most accurate and widest-ranging of any numeric type understood by the math coprocessor, and there are subtle dangers involved in using it. The word "temporary" as used in Intel's math coprocessor is quite apt: whether you use type **Extended** in your programs or not, the math coprocessor has the type available to temporarily store intermediate values that might not be fully expressible in type **Double**. If you do a lot of calculations using enormous values in variables of type **Extended**, *the math coprocessor no longer has a larger type to use to tuck away intermediate values.* In other words, if during a calculation an intermediate result appears that is larger that 10^{4932}, the math coprocessor simply doesn't have any way to express it, and a numeric overflow occurs. Your calculation will be inaccurate in range or precision or both. *Turbo Pascal will not generate an error condition to tell you that anything wrong has occurred.*

Intermediate Results and Subexpression Promotion

This may be further understood in terms of what happens "behind the scenes" during calculations of very involved mathematical expressions. The Turbo Pascal compiler evaluates expressions from left to right, "promoting" the type of the intermediate result to a larger numeric type as it goes. For example, consider this code snippet:

```
VAR
  RS : Single;
  RD : Double;

RS := 9.144E35; RD := 8.66543E255;
RS := ((RS*RS)*RD)/9.95E306;
```

The first portion of the expression to be evaluated is the subexpression **(RS*RS)**. Because the intermediate result generated by evaluating this subexpression has a magnitude of 10^{70}, it is well beyond the range of type **Single**, which has a maximum positive magnitude of 10^{38}. Even though both variables in the subexpression are of type **Single**, the compiler promotes the intermediate value to type **Double**, whose maximum positive range of 10^{308} can comfortably handle it.

Having evaluated the subexpression **(RS*RS)**, the compiler moves to the right and multiplies that intermediate result by the value of **Double** variable **RD**, which, at this point, has a magnitude of 10^{255}. The new intermediate result has a magnitude of 10^{326}. This is beyond the 10^{308} expressible by type **Double**, so the compiler promotes the intermediate result to type **Extended**. Finally, the intermediate result is divided by a literal constant with a magnitude of 10^{306}. This reduces the magnitude of the intermediate result to 10^{20}, which is within the range of type **Single**.

Now, what would have happened had there not been a type **Extended** for the intermediate result to be promoted to during the evaluation of this expression? Turbo Pascal would have assigned the pseudovalue **INF** to the expression. This pseudovalue would have carried through the evaluation and been assigned to **Single** variable **RS**, even though **RS** had enough range to accommodate the final result. Furthermore, whereas **INF** is a defined and legal IEEE floating point value, it is a difficult thing to respond to in ordinary arithmetical calculations.

Note that the general principles of promoting intermediate results as required applies to integer as well as floating point expressions. The lesson in the previous example is that in expressions containing variables of type **Extended**, there is nothing larger to store intermediate results in case of an overflow to IEEE infinity. Use **Extended** with extreme care, not only in writing expressions to minimize the "explosion" of intermediate results, but also to keep an eye on the values taken on by **Extended** variables at runtime.

Math Coprocessor Emulation (Version 5.0)

Turbo Pascal 4.0 required a math coprocessor chip to make use of the IEEE floating point real number types. Starting with release 5.0, this is no longer the case. Turbo Pascal 5.0 *emulates* the math coprocessor chip entirely in software, allowing you to run software that uses the IEEE types whether you have an 8087/287/387 chip installed in your machine or not.

Like many compromises, it's a good news/bad news situation. The good news is that having a math coprocessor allows the IEEE types to be used with maximum speed. The bad news is that emulating the math coprocessor makes for considerably slower real-number operations. It is slower, in fact, than using the old-fashioned 6-byte type **Real**.

Everything connected with floating point operations turns on what sort of code is generated by the compiler to handle floating point work. Basically, there are three kinds of floating point code that can be generated. Which kind of code is generated depends on what real number types you use and on the state of the **$N** and **$E** compiler directives.

SOFTWARE REALS

This is the code generated to support type **Real**, regardless of whether you have a math coprocessor installed, and regardless of the state of the **$N** or **$E** compiler directives.

HARDWARE IEEE REALS

The code for hardware IEEE real number support requires the presence of a math coprocessor. Instructions specific to the 8087 coprocessor are generated directly and used directly. This is the fastest and most compact floating point support provided by Turbo Pascal. You can select it by asserting **$N+** and **$E−**, which means that IEEE types are supported (**$N+**) but that math coprocessor emulation is turned off (**$E−**). Don't use it unless you *know* that your generated code will *always* be run on a machine with a math coprocessor. Executing 8087-specific opcodes on a machine without an 8087 will almost always send the machine into the bushes.

EMULATED IEEE REALS

Emulation lets you have it both ways. For each 8087-specific instruction, Turbo Pascal links in an emulation routine that mimics that instruction's actions using ordinary 8086/286/386 instructions. When you run a program containing emulated IEEE real number support, the runtime library queries the machine at runtime for the presence of a math coprocessor. If no coprocessor is found, the emulation routines do the work of the coprocessor, at some cost in speed. If a math coprocessor *is* found, however, the emulation routines are "short circuited" and the 8087-specific instructions are used instead. Therefore, if your target machine has a coprocessor, your math code will run nearly as fast as with pure hardware real number support (minus some overhead for making the decision to emulate or not). Otherwise, your math will be IEEE compatible, but it will be relatively slow. Emulation is selected by asserting **$N+** and **$E+**.

Selecting Your Floating Point Support

Floating point support is controlled by the **$N** and **$E** compiler directives. The default state is **$N−** and **$E+**. The **$N** directive is unchanged from Turbo Pascal 4.0: **$N+** allows you to use the IEEE floating point types, whereas **$N−** will limit you to type **Real**.

The **$E** directive is new to version 5.0. It stands for Emulation. **$E+** enables floating point emulation, and **$E−** disables it. **$E** defers to **$N**, in that if **$N−** is in force, the IEEE types may not be used, even if emulation is enabled with **$E+**. The different combinations of **$N** and **$E** are summarized here:

- Use **$N−** and **$E−** to force the use of type **Real**. Math coprocessors are unnecessary and are ignored.
- Use **$N+** and **$E−** for pure hardware IEEE real support. This mode *requires* the math coprocessor.
- Use **$N+** and **$E+** for emulated IEEE real support. This will allow you to use the IEEE types whether or not a math coprocessor is installed in the target machine.

Manipulating Real Numbers

Turbo Pascal includes a number of built-in mathematical and trig functions that return real number values. These are discussed at length in Chapter 16.

All the arithmetic operators, +, −, *, and /, may be used with all floating point numeric types (**Real**, **Single**, **Double**, and **Extended**). The integer division operators **DIV** and **MOD** may *not* be used with floating point types. (See Section 11.2 for a complete discussion of arithmetic operators.) Integers and floating point numbers may be freely mixed within expressions; however, the result of an expression containing a floating point number is always a floating point value and must be assigned to a floating point variable:

```
VAR
  Radius    : Integer;
  Area,PI   : Real;

PI := 3.14159;
Area := PI * Radius * Radius;
```

Here, **PI** (real) and **Radius** (integer) exist peaceably in the same expression, as long as the result of the expression is assigned to **Area**, a real number.

Coprocessor-Independent Compilation with Turbo Pascal 4.0

Because the IEEE types **Single**, **Double**, and **Extended** depend completely on the math coprocessor, the Turbo Pascal 4.0 compiler will refuse to compile programs incorporating these types if it does not detect an installed coprocessor on the system. (Version 5.0 emulates the math coprocessor if one is not installed, so this limitation does not apply to 5.0 and later releases.) This implies that you need to create separate source code files for each 4.0 application: one to support the math coprocessor and one to use the software-based type **Real**.

Not so! Turbo Pascal 4.0's conditional compilation feature makes it possible to have one source file compile to two separate .EXE files based on whether or not the compiler detects a coprocessor in the system.

I will be covering conditional compilation fully in Section 29.2. Because the following discussion assumes some familiarity with conditional compilation, you might want to read Section 29.2.

If the compiler detects a math coprocessor in the system when it begins running, it defines a metavariable named **CPU87**. You can test for whether or not **CPU87** is defined and compile sections of source code if it is or different sections if it is not.

The easiest way to have your source code both ways is to define the name of a separate floating point type. Then, define that type as being **Real** if **CPU87** is *not* defined, or as one of the IEEE types **Single**, **Double**, or **Extended** (your choice) if it *is* defined:

```
{$IFDEF CPU87}

TYPE      {This statement compiles if a coprocessor is found}
  Float = Double;

{$ELSE}
TYPE      {This one compiles if a coprocessor is NOT found}
  Float = Real;

{$ENDIF}
```

With this mechanism in place, use only type **Float** in your program to handle floating point values. Type **Float** will be a 6-byte, software-only type identical to **Real** if no coprocessor is found, and 8-byte IEEE type **Double** if a coprocessor is found.

You will note that the question still exists of distributing an .EXE file containing hardware coprocessor instructions to people who may or may not have a coprocessor installed on their machines. Unlike version 5.0, there is nothing built into the version 4.0 runtime library that detects a coprocessor at *runtime*. The cleanest solution is to upgrade to 5.0 and make use of floating point emulation.

Coprocessor-dependent floating point types in program or subprogram blocks *will hang the system* if there is no math coprocessor installed to execute the floating point instructions that manipulate the floating point values. So, to be completely safe, use type **Real** instead of one of the IEEE types **Single**, **Double**, or **Extended**, unless you can guarantee that a version 4.0 .EXE file will never be run on a machine without a coprocessor.

Internal Representation of Real Numbers

By and large, floating point numbers should be considered "black boxes" and should not be manipulated beneath the level of Turbo Pascal. Encoding analog values in a digital fashion is complicated and subtle, and it's best to let the runtime code handle everything. Exceptions may occur when you need to pass floating point values to an external assembly-language subprogram. If you're sharp enough to get that right, you're probably sharp enough to understand floating point values on a bit level.

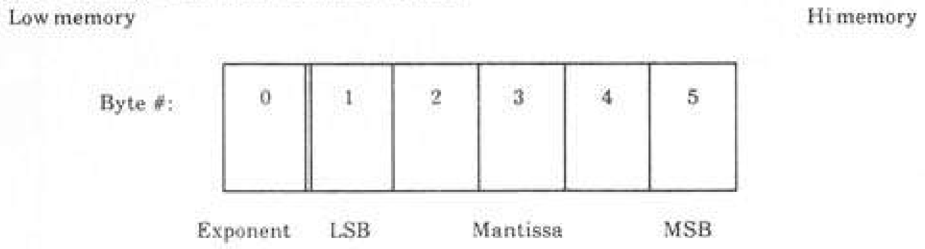
I'm not going to describe the IEEE floating point formats in detail here. Some minimal explanation is given in the Turbo Pascal 4.0 and 5.0 owner's handbooks. They are quite standard and thoroughly described in other publications.[1]

Because the 6-byte **Real** format is specific to Turbo Pascal and not used elsewhere, it's worth some description.

[1]Richard Startz, *8087 Applications and Programming for the IBM and Other PCs* (New York: Brady Co., 1983)

Figure 8.2

6-Byte Floating Point Real Representation



From low memory to high: The exponent is stored in the first byte, followed by the least significant byte of the mantissa, the next most significant byte, and so on, for five bytes of mantissa. Forty-bits-worth of mantissa will give you eleven significant figures. Beyond that point, additional precision will be lost, although the exponent will always give you the correct order of magnitude to the limits of the range of type **Real**.

Real values actually have two signs. The most significant bit of the mantissa (bit 7 of the most significant byte of the real number) is the traditional sign bit, which indicates to which side of zero the number lies. A value of 1 indicates that the number is negative.

The other sign is the sign of the exponent, which indicates to which direction the decimal point moves when converting from scientific notation to decimal notation. The value of the exponent is offset by $80. Exponents greater than $80 indicate that the decimal point must move rightward. Exponents less than $80 indicate that the decimal point must move leftward. Therefore, $80 must be subtracted from the exponent byte before the exponent can be evaluated.

Values of type **Real** in Turbo Pascal are represented internally as base 2 logarithms. In other words, to represent the number 4,673,450, Turbo Pascal stores the exponent to which the number 2 must be raised to yield 4,673,450. This makes multiplying and dividing real numbers much easier on the compiler, because numbers may be multiplied by adding their logarithms and divided by subtracting their logarithms.

The representation of type **Real** in memory is shown in Figure 8-2.

8.7: COMPUTATIONAL "REALS"

Turbo Pascal's suite of IEEE-compatible types contains one additional type which is functionally an integer but is often lumped in with the various IEEE real number types as a "computational real." This is type **Comp**. It has a range of -9.2×10^{18} to 9.2×10^{18}.

While it is true that **Comp** doesn't take a decimal part, as true floating point numbers do, it acts more like a floating point type than an integer type in several important ways. First of all, it does not work with integer disivion operators **MOD** and **DIV**. You must use the "/" operator to perform division on a variable of type **Comp**. Second, its default display format is the exponential format used by floating point numbers. To display a **Comp** variable with neither exponential notation nor unused and unusable decimal places, you must format it as you would a real number. For example:

```
VAR
  BigNum : Comp;

BigNum := 17284;
Writeln(Bignum);        { Displays as 1.72840000000000E+0004 }
Writeln(BigNum:7);      { Displays as 1.7E+0004 }
Writeln(BigNum:7:2);    { Displays as 17284.00 }
Writeln(BigNum:7:0);    { Displays as 17284 }
```

Those little peculiarities aside, **Comp** can be very useful in expressing large integer quantities, particularly financial quantities that have to be penny-accurate up to and beyond Gross-National-Product level. You must remember to express all **Comp** dollar quantities as pennies, because a **Comp** quantity may take no decimal part. Even counting pennies, **Comp** is capable of expressing dollar figures up to ten quadrillion dollars ($10,000,000,000,000,000.00), which should carry us through most reasonable financial calculations.

9 Data Structures

Allowing the programmer to build structures of data out of simple data types was perhaps the greatest advance made by Pascal over earlier computer languages like FORTRAN and Algol.

We've covered simple data types like **Integer** and **Boolean** and user-defined subranges and enumerated types. Subranges involve building down from a simple type. Now, let's take a look at building up larger structures from the same simple types.

9.1: SETS

Sets are collections of elements picked from simple types. An element either is in a set or is not in a set. The letters A, Q, W, and Z may be taken together as a set of characters. Q is in the set and L is not.

Expressed in Pascal's notation:

```
VAR
  CharSet : SET OF Char;

CharSet := ['A','Q','W','Z'];
```

A pair of square brackets when used to define a set is called a "set constructor."

Is this useful? Very. For example, sets in Pascal provide an easy way to sift valid user responses from invalid ones. In answering even a simple yes-or-no question, a user may, in fact, type two equally valid characters for yes, and two for no: Y/y and N/n. Ordinarily, you would have to test for each one individually:

```
IF (Ch='Y') OR (Ch='y') THEN DoSomething;
```

With sets, you could replace this notation with:

```
IF Ch IN ['Y','y'] THEN DoSomething;
```

The operator **IN** checks only if **Ch** is present in the set. This method not only is shorter and more readable than using ordinary equality tests, but it actually takes less machine time to execute.

In Turbo Pascal, a set type may be defined for any simple type having 256 or fewer individual values. Type **Char** qualifies, as does **Byte**. The enumerated type **Spectrum**, which we created in the last section, also qualifies, because it has only seven separate values:

```
VAR
  LowColors : SET OF Spectrum;

LowColors := [Red,Orange,Yellow];
```

Type **Integer**, however, does not. There are 65,535 different values available in type Integer, and a set may only be defined for "base types" having 256 or fewer individual values. If you define an integer subrange spanning 256 or fewer values, you may define a set with that subrange type as the set's base type:

```
TYPE
  ShoeSizes    = 5..17;

VAR
  SizesInStock : SET OF ShoeSizes;
```

You may also assign a range of elements to a set, assuming the elements are of an acceptable base type:

```
VAR
  Uppercase, Lowercase,
  Whitespace, Controls : SET OF Char;

Uppercase  := ['A'..'Z'];
Lowercase  := ['a'..'z'];
Controls   := [Chr(1)..Chr(31)];
Whitespace := [Chr(9),Chr(10),Chr(12),Chr(13),Chr(32)];
```

This is certainly easier than explicitly naming all the characters from A to Z to assign them to a set. A range of elements containing no gaps (e.g., A through Z) is called a "closed interval." The list of members within the set constructor can include single elements, closed intervals, and expressions that yield an element of the base type of the set. These must all be separated by commas, but they do not have to be in any sort of order:

```
GradeSet := ['A'..'F','a'..'f'];
BadChars := [Chr(1)..Chr(8),Chr(11),Chr(X+4),'Q','x'..'z'];
NullSet  := [];
```

You should take care that expressions do not yield a value that is outside the range of the set's base type. If **X** in **BadChars** grows to 252 or higher, the result of the expression **Chr(X+4)** will no longer be a legal character. The results of such an expression will be unpredictable, other than to say they won't do you very much good.

Sets like **Uppercase**, **Lowercase**, and **Whitespace** defined above can be very useful when manipulating characters coming in from the keyboard or other unpredictable source:

```
FUNCTION CapsLock(Ch : Char) : Char;

BEGIN
  IF Ch IN Lowercase THEN CapsLock := Chr(Ord(Ch)-32)
```

```
    ELSE CapsLock := Ch
END;


FUNCTION DownCase(Ch : Char) : Char;

BEGIN
  IF Ch IN Uppercase THEN DownCase := Chr(Ord(Ch)+32)
    ELSE DownCase := Ch
END;


FUNCTION IsWhite(Ch : Char) : Boolean;

BEGIN
  IsWhite := Ch IN WhiteSpace
END;
```

All three of these routines assume that **Uppercase**, **Lowercase**, and **Whitespace** have already been declared and filled with the proper values. Actually, the way to ensure that this is done is to do it *inside* each routine by the use of set constants, as we will do in Section 9.6. **CapsLock** returns all characters passed to it in upper case. **DownCase** returns all characters passed to it as lower case. **IsWhite** returns a **True** value if the character passed to it is "whitespace," that is, a tab, a carriage return, a linefeed, or a space character.

The **IN** operator we used above is not the only operator you may use with sets. There are two classes of set operators: operators that build sets from other sets, and operators that test relationships between sets and yield a Boolean result.

The set builder operators are:

+	Union of two sets; all elements in both sets.
*	Intersection of two sets; all elements that are present in both sets.
–	Exclusion ("set difference" in the *Turbo Pascal Reference Guide*), which yields a set of the elements in the set on the right, once all the elements in the set on the left have been removed from it. This is a tricky notion; see Section 11.3.

The set relational operators are:

IN True	if the given element is present in the set.
= True	if both sets contain *exactly* the same elements.
<> True	if the two given sets do not contain exactly the same elements.
<= True	if all the elements in the set on the left are present in the set on the right.
>= True	if all the elements in the set on the right are present in the set on the left.

More details on these set operators, including examples, will be given in Section 11.3, Set operators.

Internal Representation of Sets

Turbo Pascal implements set types as bitmaps. The memory required by any given set type depends on the number of elements in that set's base type. Because no set may have more than 256 elements, the largest legal set type will be 32 bytes in size (32 bytes × 8 bits = 256 possible set elements). Unlike most Pascal implementations, sets whose base types have fewer elements will be smaller.

I have to emphasize here that all sets of a given set type will *always* be the same size in memory. This figure is set at compile-time and does not change as elements are included in or excluded from a set.

The size in memory of a given set type will be:

```
(Max DIV 8) - (Min DIV 8) + 1
```

where **Max** and **Min** are the ordinal values of the largest and smallest items in the set. A set of **Char**, for example, runs from character 0 to character 255:

```
(255 DIV 8) - (0 DIV 8) + 1

31 - 0 + 1

32 bytes occupied by a set of Char
```

Now, consider a set with a smaller base type:

```
TYPE
  Printables = ' '..'^';
  SymbolSet = SET OF Printables;
```

Here, **Min** will be 32 (**Ord(' ')**) and **Max will be 126 (Ord('^')):**

```
(126 DIV 8) - (32 DIV 8) + 1

15 - 4 + 1

12 bytes occupied by set type SymbolSet;
```

Each possible set element has a bit in the bitmap. If a particular element is present in the set, its bit is set to binary 1; otherwise its bit remains binary 0.

The set **Uppercase** defined on page 81 is a set of **Char**. Its base type (**Char**) has 256 different elements. The set must then be represented in memory as the following sequence of 32 hex bytes:

```
00 00 00 00 00 00 00 00 FE FF FF 07 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
     Low memory      -->     High memory
```

This is a bitmap of the 256 possible elements in a set of Char. The bit that represents A is bit 1 of byte 8. The bit representing Z is bit 2 of byte 11. (Both bits and bytes are numbered from 0.) This might seem confusing at first, because the bytes are numbered from left to right while the order of the bits within each byte is numbered from right to left. If it still seems strange, write out the sequence of bytes from 8 to 11 as binary patterns, but with the bits reading from left to right instead of right to left:

```
              Byte #8 of set Uppercase

Hex FE:   11111110          01111111
          ^      ^          ^      ^
Bits:     7      0          0      7

Read:   Right-to-left    Left-to-right
```

If all the bytes in the set were expressed as binary patterns reading from left to right, the set could be written as a true bitmap.

This information on the internal representation of sets will not be especially useful to you until you need to pass a set variable to a machine language subroutine. Then, you will need to know exactly which binary bit corresponds to which element in the set, so that your machine language routine will be manipulating the correct elements of the set it receives from your Pascal program.

9.2: ARRAYS

An array is a data structure consisting of a fixed number of identical elements, with the whole collection given a single identifier as its name. The program keeps track of array elements by number. Sometimes you name the entire array to work with it. Most of the time, you identify one of the individual elements, by number, and work with that element alone. This number identifying an array element is called an "index." In Pascal, an index need not be a traditional number. Enumerated types, characters, and subranges may also act as array indices, allowing a tremendous richness of expression not matched by any other computer language.

It may be helpful to think of an array as a row of identical empty boxes in memory, side by side. The program allocates the boxes, but it is your job as programmer to fill them and manipulate their contents. The elements in an array are, in fact, set side-by-side in order in memory.

An array element may be of any data type except a file. Arrays may consist of data structures; you may have arrays of records and arrays of arrays. An array index must be a member or a subrange of an ordinal type, or a programmer-defined enumerated type. Floating point numbers may *not* act as array indices, nor may **LongInt** or **Comp**. Type **Integer**, **ShortInt**, **Byte**, **Word**, **Char**, and **Boolean** may all index arrays.

Here are some valid array declarations, just to give you a taste of what is possible (if you don't understand for now what a "record" is, bear with us for the time being, or look ahead to Section 9.3):

```
CONST
  Districts   = 14;

TYPE
  String80    = String[80];
  Grades      = 'A'..'F';                         { Subrange    }
  Percentile  = 1..99;                            { Ditto       }
                                                  { Enum. type  }
  Levels      = (K,G1,G2,G3,G4,G5,G6,G7,G8,G9,G10,G11,G12);
                                                  { Ditto       }
  Subjects    = (English,Math,Spelling,Reading,Art,Gym);

  Profile     = RECORD
                  Name     : String80;
                  SSID     : String80;
                  IQ       : Integer;
                  Standing : Percentile;
                  Finals   : ARRAY[Subjects] OF Grades
                END;

  GradeDef   = ARRAY[Grades] OF String80;

VAR
  K12Profile : ARRAY[Levels] OF Profile;
  Passed     : ARRAY[Levels] OF Boolean;
  Subtotals  : ARRAY[1..24] OF Integer;
  AreaPerc   : ARRAY[1..Districts] OF Percentile;
  AreaLevels : ARRAY[1..Districts] OF ARRAY[Levels] OF
                Percentile;
  RoomGrid   : ARRAY[1..3,Levels] OF Integer;
```

The declarations shown above are part of an imaginary school district records manager program written in Pascal. Note that **Passed** is an array whose index is an enumerated type. **Passed[G5]** would contain a Boolean value (**True** or **False**) indicating whether a student had passed or failed the fifth grade. Remember, **G5** is *not* a variable; it is a constant value, one value of an enumerated type.

The low limit and high limit of an array's index are called its *bounds. The bounds of an ordinary array in Pascal must be fixed at compile time.* The Pascal compiler must know when it compiles the program exactly how large all data items are going to be. You cannot **REDIM** an ordinary array as you might in BASIC, nor can you change its shape as you might in APL.

With this in mind, the variable **AreaPerc** deserves a closer look. At first glance, you might think it has a variable for a high bound, but, actually, **Districts** is a constant with a value of 14. Writing **[1..Districts]** is no different from writing **[1..14]**, but within the context of the program, it assists your understanding of what the array actually represents.

Most of the arrays shown in the above example are one-dimensional. A one-dimensional array has only one index. A two-dimensional array (e.g., **RoomGrid**, page 85) has two indices. An array may have any number of dimensions, but it edges toward bad practice to define an array with more than two or, on the outside, three (Figure 9.1). Also, the more dimensions an array has, the larger it tends to be—and the more likely that parts of it are empty or full of duplicate or rarely accessed information that does nothing for you but waste memory. There are better ways, less wasteful of memory, to handle large, complicated data structures, such as pointers.

When Turbo Pascal allocates an array in memory, it does not zero out the elements of the array, as BASIC would. In general, *Pascal does not initialize data items for you.* If there is garbage in the area of RAM in which the compiler sets up an array, the array elements will contain the garbage when the array is allocated. If you wish to zero out or otherwise initialize the elements of an array, you must do it yourself, in your program, before you use the array. This is not difficult. Use a **FOR** loop (see Section 13.4 for more on **FOR** loops):

```
FOR I := 1 TO 24 DO Subtotals[I] := 0;
```

If portability is not a consideration, arrays (especially *large* arrays) may be initialized even more quickly with Turbo Pascal's **FillChar** statement:

```
FillChar(Subtotals,SizeOf(Subtotals),Chr(0));
```

Details on using **FillChar** can be found in Section 23.4.

Good examples of arrays used effectively can be found in the **ShellSort** procedure in Section 14.2 and the **QuickSort** procedure in Section 14.4.

Several special array identifiers are defined in Turbo Pascal specifically for treating all of memory as one large array. **Mem** treats memory as an array of **Byte**, **MemW** treats memory as an array of **Integer** or **Word**, and **MemL** treats memory as an array of **LongInt**. The indices of these special arrays must include both an 8086 segment and an offset, separated by a colon. I'll describe them in detail in Section 23.3.

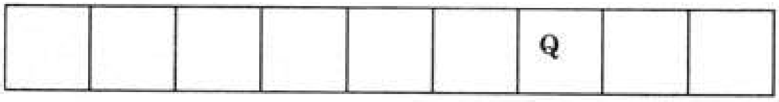
Internal Representation of Arrays

Arrays are sequences of the same data type. Turbo Pascal allocates array elements from low memory to high. In other words, the first byte in the first element of an array has the lowest address of any byte in the array.

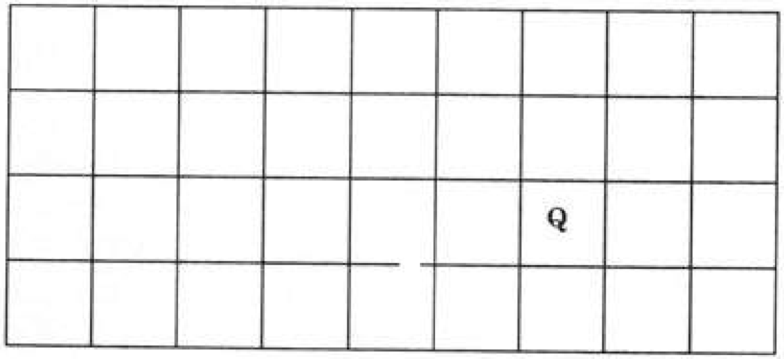
For multidimensional arrays, the elements are stored with the rightmost dimension increasing first. This is best shown as a diagram (Figure 9.2).

Figure 9.1

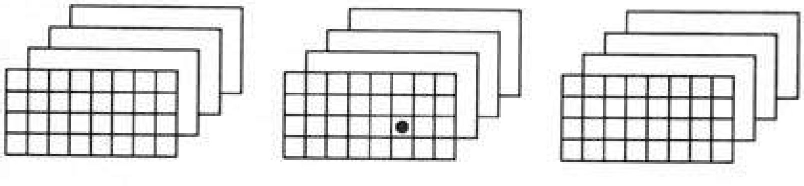
Multidimensional Arrays



D1 : ARRAY[0..8] OF CHAR;
D1[6] contains the "Q"



D2 : ARRAY[0..8,1..4] OF CHAR;
D2[6,3] contains the "Q"



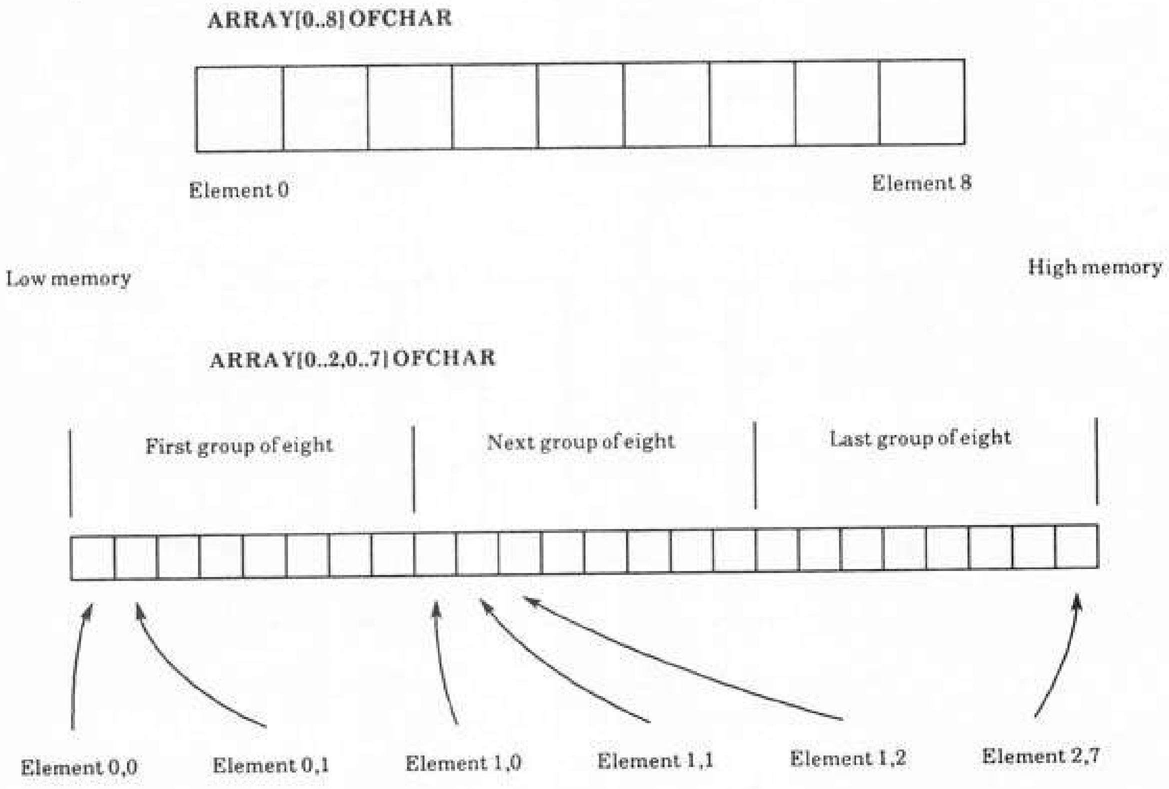
D4 : ARRAY[0..7,1..4,0..3,0..2] OF CHAR;
D4[5,3,0,1] contains the "Q"

9.3: RECORDS

An array is a data structure composed of a number of identical data items all in a row and referenced by number. This sort of data structure is handy for dealing with large numbers of the same type of data, such as values returned from an experiment of some

Figure 9.2

Arrays in Memory



Multidimensional arrays are stored in memory with the rightmost dimensions increasing first.

sort. You might have a collection of 500 temperature readings, and you need to average them and perform analysis of variance on them. The easiest way to do that is to load them into an array and work with the temperature readings as elements of the array.

There is a data structure composed of data items that are *not* of the same type. It is called a "record," and it gets its name from its origin as one line of data (one record) in a data file.

A record is a structure composed of several data items of different types grouped together. These data items are called the "fields" of the record. An auto repair shop might keep a file on its spare-parts inventory. For each part they keep in stock, they need to record its part number, its description, its wholesale cost, retail price, customary stock level, and current stock level. All these items are intimately linked to a single physical gadget (a car part). To simplify their programming, the shop puts the fields together to form a record:

```
TYPE
  PartRec  =  RECORD
                 PartNum, Class   : Integer;
                 PartDescription : String;
                 OurCost          : LongInt;
                 ListPrice        : LongInt;
                 StockLevel       : Integer;
                 OnHand           : Integer;
              END

VAR
  CurrentPart, NextPart  : PartRec;
  PartFile      : FILE OF PartRec;
  CurrentStock : Integer;
  MustOrder     : Boolean;
```

The entire structure becomes a new type with its own name. Data items of the record type can then be assigned, written to files, and otherwise worked with as a single entity without having to explicitly mention all the various fields within the record.

```
CurrentPart := NextPart;   { Assign part record to another }
Read(PartFile,NextPart);   { Read next record from file    }
```

When you need to work with the individual fields within a record, the notation consists of the record identifier followed by a period followed by the field identifier:

```
CurrentStock := CurrentPart.OnHand;
IF CurrentStock < CurrentPart.StockLevel
  THEN MustOrder := True;
```

Accessing individual fields within a record this way is sometimes called "dotting."

Relational operators may *not* be used on records. To say that one record is "greater than" or "less than" another cannot be defined since there are an infinite number of possible record structures with no well-defined and unambiguous order for them to follow. In the example above we are comparing the *fields* of two records, not the records themselves. The fields are both integers and can therefore be compared by the < operator.

The WITH Statement

Fields within a record are accessed by dotting:

```
CurrentPart.OurCost := 1075;      { In pennies! }
CurrentPart.ListPrice := 4185;    { " }
CurrentPart.OnHand := 4;
```

There is some unnecessary repetition here. If you have to go down a list of fields within the same record and work with each field, you can avoid specifying the identifier of the record each and every time by using a special statement called the **WITH** statement.

We could simplify assigning values to several fields of the same record by writing the above snippet of code this way:

```
WITH CurrentPart DO
  BEGIN
    OurCost := 1075;
    ListPrice := 4185;
    OnHand := 4;
  END;
```

The space between the **BEGIN** and **END** is the *scope* of the **WITH** statement. Within that scope, the record identifier need not be given to work with the fields of the record named in the **WITH** statement. (If you are very new to Pascal, you might return to this section after reading the general discussion on statements in Chapter 12. **WITH** statements are subject to the same rules that all types of Pascal statements obey.)

WITH statements need not have a **BEGIN/END** unless they contain more than a single statement. A **WITH** statement may include only a single statement to work with a record if that single statement contains several references to fields within a single record:

```
WITH CurrentPart DO VerifyCost(OurCost,ListPrice,Check);
```

In this example, **VerifyCost** is a procedure that takes as input two price figures and returns a value in **Check**. (Procedures are covered fully in Chapter 14.) Without the **WITH** statement, calling **VerifyCost** would have to be done this way:

```
VerifyCost(CurrentPart.OurCost, CurrentPart.ListPrice,
Check);
```

The **WITH** statement makes the statement crisper and easier to understand.

Nested Records

A record is a group of data items taken together as a named data structure. Whereas a record is itself a data item, records themselves may be fields of larger records. Suppose the repair shop we've been speaking of expands its parts inventory so much that finding a part bin by memory gets to be difficult. There are 10 aisles with letters from A through J, with the bins in each aisle numbered from 1 up. To specify a location for a part requires an aisle character and a bin number. The best way to do it is by defining a new record type:

```
TYPE
  PartLocation = RECORD
                   Aisle : 'A'..'J';
                   Bin   : Integer
                 END;
```

Since each part has a location, **PartRec** needs a new field:

```
TYPE
  PartRec   = RECORD
                PartNum, Class  : Integer;
                PartDescription : String;
                OurCost         : LongInt;
                ListPrice       : LongInt;
                StockLevel      : Integer;
                OnHand          : Integer;
                Location        : PartLocation { A record! }
              END;
```

Location is nested within the larger record. To access the fields of **Location** you need *two* periods:

```
LookAisle := CurrentPart.Location.Aisle
```

If the outermost record is specified by a **WITH** statement, you might have an equivalent statement like this:

```
WITH CurrentPart DO LookAisle := Location.Aisle;
```

WITH statements are fully capable of handling many levels of record nesting. You may, first of all, nest **WITH** statements one within another. The following compound statement is equivalent to the previous statement:

```
WITH CurrentPart DO
  WITH Location DO LookAisle := Aisle;
```

The **WITH** statement also allows a more concise form to express the same thing:

```
WITH CurrentPart, Location DO LookAisle := Aisle;
```

For this syntax, you must place the record identifiers after the **WITH** reserved word, separated by commas, *in nesting order.* That is, the name of the outermost record is on the left, and the names of records nested within it are placed to its right, with the innermost nested record placed last.

Records are used most often as "slices" of a disk file. We will explore this use of records much more fully in the general discussion of binary file I/O in Section 19.8.

9.4: VARIANT RECORDS

In the previous section, we looked at Pascal's way of grouping several different data types together and calling them a record. In a "fixed" record type, which we have been discussing, the fields that make up a record are always the same data items in the same order and never change.

In a program doing real work in the real world, a record usually represents some real entity—and the real world is a varied and messy place. Pascal makes a major concession to the messiness of the real world by allowing "variant records," the fields of which may be different data types depending on the contents of the other fields.

The notion of variant records is a subtle one. You might wish to come back to this section after reading the rest of Part Two, having paid particular attention to the discussion of **CASE OF** statements in Section 13.3. The definition of every variant record contains a **CASE OF** construct, and without a reasonable understanding of **CASE OF**, you will not understand how variant records work.

Let's continue with the example of the auto repair shop's parts-inventory system. In their system, each part that they keep in stock has a record in a file containing price information, stock levels, and a location in their storage room. However, in some cars there are parts that break so rarely that they are practically never needed, and it would be financially foolish to keep such parts in stock.

Moreover, if one such part breaks, the shop must have some means of ordering the part quickly. It is necessary to store (for those parts only) a vendor name and phone number for emergency ordering and a suspected lead time on the order based on previous discussions with the vendor. If the part has to come all the way from Japan, it is better if the customer knows about it up front, rather than having him haunt the repair shop for weeks while the part is in shipment.

We can see, then, that there are two types of parts: those kept in stock and those obtainable via emergency order only. Both types have a part number, a class, a description, and a cost value. But, there the similarity ends. We might have two separate record definitions, one for each type of part:

```
TYPE
  STPartRec   = RECORD
                  PartNum, Class     : Integer;
                  PartDescription    : String;
                  OurCost            : LongInt;
                  ListPrice          : LongInt;
                  StockLevel         : Integer;
                  OnHand             : Integer;
                  Location           : PartLocation
                END;


EOPartRec   = RECORD
                  PartNum, Class     : Integer;
                  PartDescription    : String;
                  OurCost            : LongInt;
                  ListPrice          : LongInt;
                  Vendor             : String;
                  OrderPhone         : String;
                  OrderLead          : Integer;
                END;
```

The new part record type handles all information for emergency-order parts quite nicely. However, as we will see in Section 19.8, a binary file may store only one record type, not two. Need we now keep two separate parts files, one for stocked parts and one for emergency-order parts?

No, the two fixed record types may be combined into a single variant record type. The new record looks like this:

```
PartRec   = RECORD
                PartNum, Class     : Integer;
                PartDescription    : String;
                OurCost            : LongInt;
                ListPrice          : LongInt;
                CASE Stocked : Boolean OF
                  True :
                    (StockLevel    : Integer;
                     OnHand        : Integer;
                     Location      : PartLocation);
                  False :
                    (Vendor        : String;
                     OrderPhone    : String;
                     OrderLead     : Integer)
              END;
```

The new record type has two distinct parts. The first part, including the fields from **PartNum** to **ListPrice**, is called the "fixed part" of the record type. The rest of the record, from the reserved word **CASE** to the end, is the "variant part" of the record type. The variant part of a variant record must always be the last part of the record. You cannot have more fixed part fields after the variant part.

The difference hinges on the **CASE** construct. The **CASE** construct provides two or more alternative field definitions based on the value of what is called the "tag field." The tag field of **PartRec** is the Boolean variable **Stocked**.

Stocked, being a Boolean variable, has only two possible values, **True** and **False**. In those **CASE**s where **Stocked** is **True** (if the record refers to a part kept in stock) the record contains the fields **StockLevel**, **OnHand**, and **Location**. In those **CASE**s where **Stocked** is **False** (for emergency-order parts), the record instead contains the fields **Vendor**, **OrderPhone**, and **OrderLead**.

Note the use of parentheses to set off the separate variant parts of a variant record. Parentheses *must* surround the field definitions of each variant part. Do not include the tag field value constants (**True** and **False** in our example) within the parentheses.

It is not just a matter of the record actually containing both variant parts and "hiding" the one not currently selected by **Stocked**. *Variant parts of a variant record that are not selected by the tag field are inaccessible, and their values are undefined.* Such fields are literally not there, and data which were previously contained in those fields vanish when the tag field changes.

You must keep this in mind to stay out of certain kinds of trouble. For example, if **Stocked** = **True** and you perform this assignment:

```
CurrentPart.OnHand := 6;
```

and then later in the program perform this assignment on that same record:

```
CurrentPart.Stocked := False;
```

CurrentPart.OnHand is now inaccessible. If you try to read the field **OnHand**, the code may present you with garbage. Furthermore, if you then assign

```
CurrentPart.Stocked := True;
```

although the field **OnHand** is now accessible, it may or may not retain its old value. It is undefined until you assign a new value to it. Turbo Pascal will *not* give you run-time errors for attempting to access fields in unselected variant parts of a variant record. It will place either garbage data or parts of other unrelated fields in the fields you access.

Although our example record type for simplicity's sake has only two variants (because the tag field is Boolean and has only two possible values), a variant record may have up to 256 different variants, and any ordinal type may act as a tag field. This allows the hazardous possibility of a tag field assuming a value for which there is no defined variant. For example, if your tag field is of type **Char** and you define variants only for cases in which the tag field contains **A**, **B**, or **C**, what happens if the tag field is assigned a value of **Z**? The variant part of the record becomes undefined, and attempts to access a field in a variant part may return garbage. *Avoid this possibility at all costs.* The best way is to make sure that *all* possible values of a tag field have a defined variant. Instead of making a tag field of type **Char**, define a subrange type

```
TYPE
  TagChar = 'A'..'C';
```

and make the tag field type **TagChar** instead. Enumerated types (see Section 7.4) are also extremely useful in creating data structures incorporating variant records.

It is perfectly legal for the variant parts of a variant record to be of different sizes. If so, how does the compiler allocate space for a variant record in memory? Every instance of a variant record type is allocated as much memory as is required by the largest variant. If data storage space is short, you might check to see if you are wasting space by defining an enormous variant of a record and then rarely (or never) using it. Every time you define a record of that type, the record will use all the space needed by that enormous variant even if that variant is never selected by the tag field. Rearranging your data structure to break out that enormous variant as a separate type could save you a lot of memory and disk space!

Free Union Variant Records

There are actually two types of variant records defined by the Pascal language, and both of them are implemented in Turbo Pascal. The variant record we have just described is called a "discriminated union," although the term is not used very much outside of academic circles. Discriminated union variant records always have a tag field. It is possible to define a variant record *without* a tag field. Such a data structure is called a "free union" variant record.

At first glance, a variant record without a tag field would appear to be a call to chaos (some experts consider it exactly that). Without a tag field to select one variant part from the many, how can the compiler know which variant part is currently in force?

The answer is easy: They are *all* in force, all at once.

It is a peculiar concept and one that should not be used without complete understanding of its implications. An example will help:

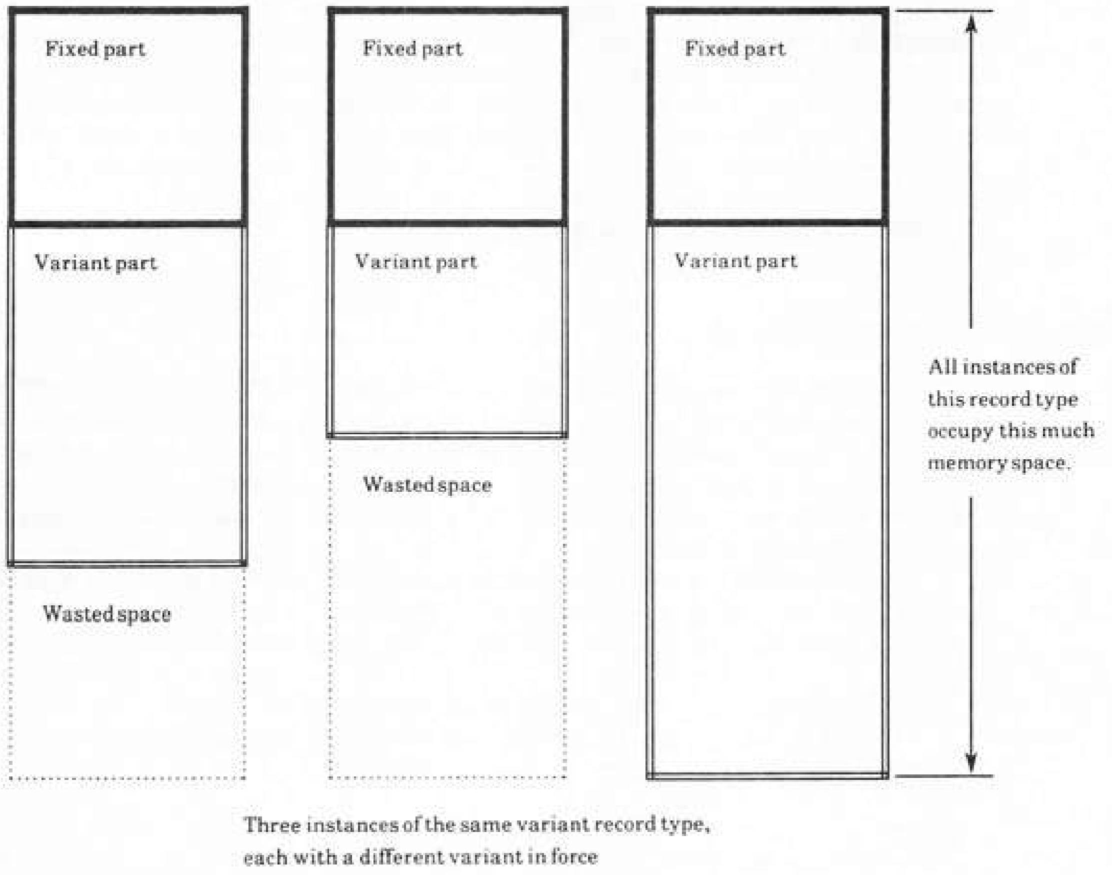
```
TYPE
  Halfer = RECORD
             CASE Boolean OF
               True  : (I : Integer);
               False : (HiByte : Byte;
                        LoByte : Byte);
           END;

VAR
  Porter : Halfer;
```

As you can see, there is no tag field present. There is, however (and this is the most peculiar thing about a free union) a tag field *type*, which, in this case, is **Boolean**. The tag field type must be present, and it must be an ordinal type with 256 or fewer values.

The tag field type is given only to specify the number of possible variant parts of the record and the values by which the variants are selected.

Figure 9.3

Variant Records


Fixed part
Variant part
Wasted space
Fixed part
Variant part
Wasted space
Fixed part
Variant part
All instances of this record type occupy this much memory space.
Three instances of the same variant record type, each with a different variant in force

You'll remember that any time the tag field changes in an ordinary discriminated union variant record, the values of the fields in the previously selected variant part go away, and the values of the fields in the newly selected variant part are undefined until some new values are assigned to those fields. To add to the confusion, Turbo Pascal does not enforce this. There *might* be some carryover in the values as variants are selected and deselected, but there might not. The Turbo Pascal compiler makes no promises.

Here, the tag field cannot change because it does not exist. Any value assigned to any field of any variant part of a free union remains, and that value will be interpreted by all the other variant parts which share that same region of memory.

To understand that, you must break a prime taboo in ISO Standard Pascal and look at the actual memory locations underlying a program's data area. All variant parts of a free union are mapped onto the same region of memory. Our free union type **Halfer** has two variant parts: One has a single **Integer** field, and the other has two **Byte** fields. We know that an integer is stored as 2 bytes in memory. Values assigned to **Porter.HiByte** occupy the same physical byte of memory in which the high-order byte of **Porter.I** exists. Values assigned to **Porter.LoByte** occupy the same physical byte of memory in which the low-order byte of **Porter.I** exists.

When we assigned a value to **Porter.I**:

```
Porter.I := 21217;
```

and then examine the value of **Porter.HiByte**, we will find that **Porter.HiByte** contains the high-order byte of **Porter.I**, in this case 82. In similar fashion, we can change the high-order byte of **Porter.I** without disturbing the value of the low-order byte, and vice versa.

What good is all this? Free unions can get you out of some tight spots. For example, some I/O devices are designed so that they can move only 1 8-bit byte at a time to an I/O port. To send a 16-bit quantity to an I/O port, you must break it down into 2 bytes. There are mathematical ways of deriving the values of the individual high and low order bytes of an integer, but they involve arithmetic, which takes time. A faster way is simply assigning your integer to a data type which can *simultaneously* be treated as both a single integer and as 2 8-bit bytes. To send an integer to an 8080 I/O port using the **Halfer** free union variant record, you would do this:

```
J := 21217;
Porter.I := J;                    { Porter is type Halfer }
Port[$A0] := Porter.HIByte;       { Out to I/O port $A0   }
Port[$A0] := Porter.LOByte;       { one byte at a time... }
```

(The details of port I/O are covered in Section 23.6.) The free union variant record is actually a trick to circumvent the strong typing restrictions Pascal places upon you. It is a way of mapping one type upon another so that any type can be converted to any other type *if you know what you are doing.* Free unions are actually a concession to the vagaries of the standardless hardware world, and should be used only to defeat problems imposed by the hardware itself.

There are more terse and less magical ways of doing this sort of thing, mainly with type casting (see Section 12.3.). Free unions are your best choice when portability is a consideration, however. They are often not portable in their effects, but are certainly more portable to other compilers than are Turbo Pascal's proprietary extensions.

9.5: STRINGS

Manipulating words and lines of text is a fundamental function of a computer program. At minimum, a program must display messages like "Press RETURN to continue:" and

"Processing completed." In Pascal, as in most computer languages, a line of characters to be taken together as a single entity is called a "string."

Standard Pascal Strings (Fixed Length)

ISO Standard Pascal has very little power to manipulate strings. In Standard Pascal, there is nothing formally referred to as type **String**. To hold text strings, Standard Pascal uses a packed array of characters, sometimes abbreviated as **PAOC**:

```
TYPE
  PAOC25 = PACKED ARRAY[1..25] OF Char;
```

This is a typical definition of a **PAOC** type. Of course, it doen't have to be 25 characters in size; it can be as large as you like. The word **PACKED** is a holdover from large mainframe computers; on 8- or 16-bit computers it serves no purpose. In mainframe computers with 32- or 64-bit words, the word **PACKED** instructs the compiler to store as many characters as will fit in one machine word, rather than using one machine word per character. Using **PACKED** on a mainframe could reduce the size of a string by a factor of four to eight. Turbo Pascal always stores a **Char** value in 1 byte no matter what the word size of the computer it runs on. The word **PACKED** is still necessary, however, to define a Standard Pascal string.

What can be done with a **PAOC**-type string? Not much; a string literal can be assigned to it if the string literal is *exactly* the same size as the definition of the **PAOC**:

```
VAR
  ErrorMsg : PAOC25;

ErrorMsg := 'Warning! Bracket missing!';   { This is OK }
ErrorMsg := 'Warning! Comma missing!';     { Illegal!   }
```

The length of the second string literal is two characters short of 25, so the compiler will display

```
Error #26: Type mismatch
```

The second string literal could not be considered a **PAOC25** because it is only 23 characters in length; hence the type mismatch error. Of course, you could have padded out the second literal with spaces, and the padded literal would have been acceptable.

You can compare two **PAOC**-type strings with the relational operators, read and write them from files, and print them to the screen. And that's where it ends. All other manipulations have to be done on a character-by-character basis, as though the string were just another array of any simple type.

The only time to use **PAOC**-type strings is in situations in which you are writing code that may have to be ported to a Pascal compiler that does not understand variable-length (sometimes called "dynamic") strings. Such compilers are rare and getting rarer. As you might expect, they seem to be found mainly on enormous mainframe computers.

Turbo Pascal Variable-Length Strings

Most modern Pascal implementors, including those who wrote Turbo Pascal, have filled this hole in the language definition by providing what are called "variable-length strings." Like **PAOC**-type strings, variable-length strings are arrays of characters, but they are treated by the compiler in a special way. Variable-length strings have a "logical length," the value of which varies depending on what you put into the string. Strings of different logical lengths may be assigned to one another as long as the real, physical lengths of the strings are not exceeded. The means of implementing variable-length strings in Turbo Pascal is identical to the way they have been implemented in Pascal/MT+and UCSD Pascal.

A string type in Turbo Pascal has two lengths: a physical length and a logical length. The physical length is the amount of memory the string actually takes up. This length is set at compile time and never changes. The logical length is the number of characters currently stored in the string. This can change as you work with the string. The logical length (which from now on we will simply call the length) is stored as part of the string itself and can be read in several ways.

A string variable is defined using the reserved word **String**. The default physical length for a string defined as **String** is 255 characters. This differs from Turbo Pascal V3.0 and earlier, in which the **String** type had no default length, and derived string types of some given size had to be declared in the **TYPE** section of your programs. You may not always need a string that physically large (for example, a telephone number fits comfortably in fewer than 20 characters); and, you may define smaller string types to save memory by placing the physical size after the reserved word **String** in brackets:

```
VAR
  Message : String[80];  { Physical size = 80 }
  Name    : String[30];  { Physical size = 30 }
  Address : String[30];  { Physical size = 30 }
  State   : String[2];   { Physical size = 2  }
```

The range of legal physical lengths is 1 to 255 characters.

You may also define separate string types as strings of different physical lengths. This is a much better way to deal with strings of fewer than 255 characters:

```
TYPE
  String80 = String[80];
  String30 = String[30];
  Buffer   = String[255];
```

Once you have defined these types, declare all string variables that are to have a physical length of 30 characters as type **String30**. This way, all such strings will have identical types and not simply compatible types. This becomes critical when you have to pass string variables as reference parameters. See Section 12.2 for a discussion of compatible and identical types, and Section 14.2 for a discussion of reference parameters.

What is a string, physically? A string is an array of characters indexed from 0 to the physical length. Character 0 is special, however; it is the "length byte" and it holds the logical length of the string at any given time. The length byte is set by the runtime code when you perform on a string an operation that changes its logical length:

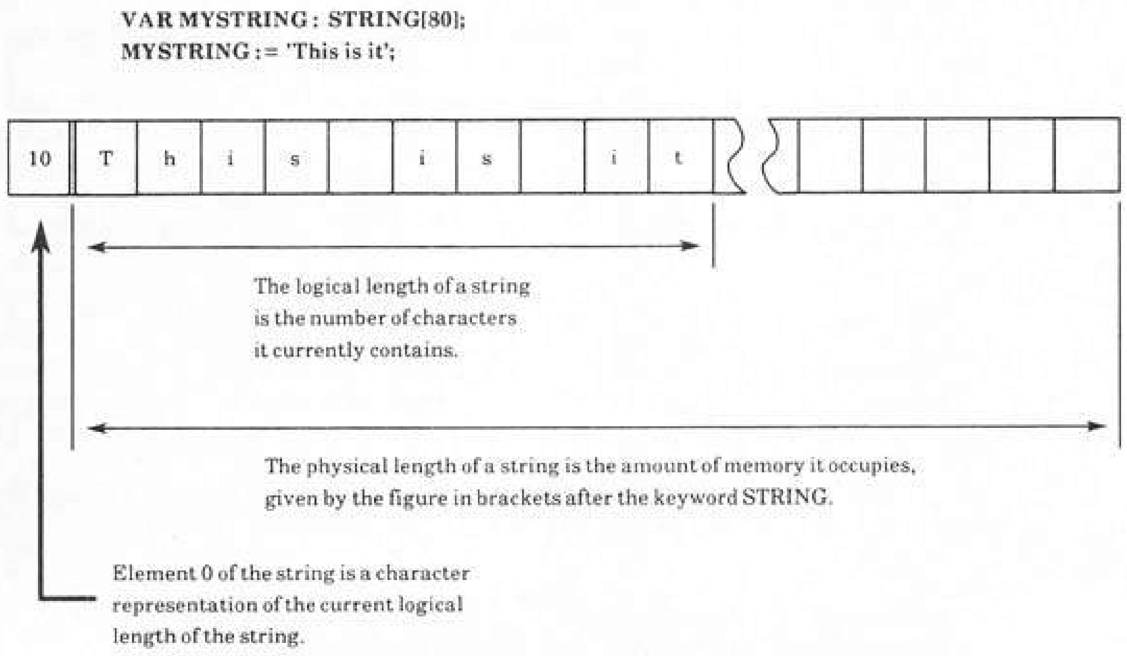
```
MyString := '';                               { MyString[0] = 0  }
MyString := 'Frodo';                          { MyString[0] = 5  }
MyString := 'Alfred E. ' + 'Newmann';         { MyString[0] = 17 }
```

The actual text contained in the string begins at **MyString[1]** (Figure 9.4).

Strings may be accessed as though they were, in fact, arrays of characters. You can reference any character in the string, including the length byte, with a normal array reference:

Figure 9.4

String Representation


VAR MYSTRING : STRING[80];
MYSTRING := 'This is it';
10
T
h
i
s
i
s
i
t
The logical length of a string is the number of characters it currently contains.
The physical length of a string is the amount of memory it occupies, given by the figure in brackets after the keyword STRING.
Element 0 of the string is a character representation of the current logical length of the string.

```
VAR
  MyString   : String[15];
  CharS      : Integer;
  OUTChar    : Char;

MyString := 'Galadriel';
CharS     := ORD(MyString[0]);  { CharS now equals 9     }
OUTChar   := MyString[6];       { OUTChar now holds 'r' }
```

Even though the Turbo Pascal runtime code treats the length byte as a number, it is still an element in an array of **Char** and thus cannot be assigned directly to type **Byte** or **Integer**. To assign the length byte to a numeric variable, you must use the **ORD** transfer function, which is Pascal's orderly way of transferring a character value to a numeric value (see Section 16.5).

This, however, is doing it the hard way. There are a good number of predefined string-handling functions and procedures built into Turbo Pascal. The **Length** function is a good example. It returns the current logical length of a string variable:

```
Chars := Length(MyString);      { Chars now equals 9     }
```

Under Turbo Pascal 4.0 (but not 5.0), **Length** has the surprising property of being a "reversible" function; you can also *set* the length of a string by placing the **Length** function on the left side of the assignment operator:

```
Length(MyString) := 5;
```

This statement sets the logical length of the string to five characters, essentially truncating it to that size regardless of what it contained before. We will discuss **Length** and all the other built-in string handling functions and procedures in Chapter 15.

Characters and strings are compatible in some limited ways. You can assign a character value to a string variable. The string variable then has a logical length of one:

```
MyString := 'A'            { Logical length = 1 }
MyString := OUTChar;       { Ditto }
```

A string (even one having a length of 0 or 1) cannot be assigned to a character variable. You can compare a string variable to a character literal:

```
IF MyString = 'A' THEN StartProcess;
```

It is possible to assign a string to another string with a shorter physical length. This will cause neither a compile time nor a run time error. What it *will* do is truncate the data from the larger string to the maximum physical length of the smaller string.

9.6: TYPED CONSTANTS

Standard Pascal allows only simple constants: Integers, characters, reals, Booleans, and strings. Turbo Pascal also provides "typed" constants, which are constants with an explicit type that are initialized to some specific value.

Calling Turbo's typed constants "constants" is not entirely fair. Real constants are hardcoded "in-line" into the machine code produced by the compiler, with a copy of the constant everywhere it is named. The constant thus exists at no single address. Turbo Pascal's typed constants are actually static variables that are initialized at runtime to values taken from the source code. They exist at one single address that is referenced any time the typed constant is used.

Typed constants also violate the most fundamental proscription of constants in all languages: They may be changed during the course of a program run. Of course, you are not obligated to alter typed constants at runtime, but the compiler will not stop you if you try. In this, Turbo Pascal provides less protection than in the area of simple constants. If you attempt to write to a simple constant, Turbo Pascal will display

```
Error 113: Error in statement
```

With that in mind, it might be better to think of typed constants as a means of forcing the compiler to initialize complicated data structures. Standard Pascal has *no* means of initializing variables automatically. If values are to be placed into variables, *you* must place them there somehow, either with assignment statements or by reading values in from a file. For example, you could initialize an array of 15 integers this way:

```
VAR
  Weights : ARRAY[1..15] OF Integer;

Weights[1]  := 17;
Weights[2]  := 5;
Weights[3]  := 91;
Weights[4]  := 111;
Weights[5]  := 0;
Weights[6]  := 44;
Weights[7]  := 16;
Weights[8]  := 3;
Weights[9]  := 472;
Weights[10] := 66;
Weights[11] := 14;
Weights[12] := 38;
Weights[13] := 57;
Weights[14] := 8;
Weights[15] := 10;
```

For 15 values this may seem manageable. But suppose you had 50 values? Or 100? At that point, Turbo Pascal's typed constants become *very* attractive. This same array could be initialized as a structured constant like so:

```
CONST
  Weights : ARRAY[1..15] OF Integer =
    (17,5,91,111,0,44,16,3,472,66,14,38,57,8,10);
```

The form of a typed constant definition is this:

```
<identifier> : <type> = <values>
```

Because Turbo Pascal allows multiple **CONST** keywords within a single program, you may have a separate **CONST** declaration section for typed constants *after* the type declaration section. This allows you to declare your own custom type definitions and then create constants of your custom types.

Array Constants

The example above is a simple, one-dimensional array constant. Its values are placed, in order, between parentheses, with commas separating the values. *You must give one value for each element of the array constant.* Turbo will not allow you to initialize some values of an array and leave the rest blank. You must do all of them or none at all. If the number of values you give does not match the number of elements in the array, Turbo will display

```
Error 87 : ',' expected.
```

It literally is looking for another comma and more values in the list.

If you need to initialize only a few values out of a large array (leaving the others undefined), it might be more effective to go back to individual assignment statements.

You may also define multidimensional array constants. The trick here is to enclose each dimension in parentheses, with commas separating both the dimensions and the items. A single pair of parentheses must enclose the entire constant. The innermost nesting level represents the rightmost dimension from the array declaration. An example will help:

```
CONST
  Grid : ARRAY[0..4,0..3] OF Boolean =
                ((4,6,2,1),
                 (3,9,8,3),
                 (1,7,7,5),
                 (4,1,7,7),
                 (3,1,3,1));
```

This is a two-dimensional array of integer constants, arranged as five rows of four columns, and might represent game pieces on a game grid. Adding a third dimension to the game, and the grid, would be done this way:

```
CONST
  Space : ARRAY[0..7,0..4,0..3] OF Integer =

(((4,6,2,1),(3,9,8,3),(1,7,7,5),(4,1,7,7),(3,1,3,1)),
 ((1,1,1,1),(1,1,1,1),(1,1,1,1),(1,1,1,1),(1,1,1,1)),
 ((2,2,2,2),(2,2,2,2),(2,2,2,2),(2,2,2,2),(2,2,2,2)),
 ((3,3,3,3),(3,3,3,3),(3,3,3,3),(3,3,3,3),(3,3,3,3)),
 ((4,4,4,4),(4,4,4,4),(4,4,4,4),(4,4,4,4),(4,4,4,4)),
 ((5,5,5,5),(5,5,5,5),(5,5,5,5),(5,5,5,5),(5,5,5,5)),
 ((6,6,6,6),(6,6,6,6),(6,6,6,6),(6,6,6,6),(6,6,6,6)),
 ((7,7,7,7),(7,7,7,7),(7,7,7,7),(7,7,7,7),(7,7,7,7)));
```

The values given for the two-dimensional array have been retained here to see how the array has been extended by one dimension. Note that the list of values in an array constant must begin with the same number of left parentheses as the array has dimensions. Remember, also, that *every* element in the array must have a value in the array constant declaration.

Notice that this feature of Turbo Pascal allows you to initialize 160 different integer values in a relatively small space. Imagine what it would have taken to initialize this array with a separate assignment statement for each array element!

Record Constants

Record constants are handled a little bit differently. You must first declare a record type and then a constant containing values for each field in the record. The list of values must include the name of each field followed by a colon and then the value for that field. Items in the list are separated by semicolons. As an example, consider a record containing configuration values for a terminal program:

```
TYPE
  BPS          = (B110,B300,B1200,B2400,B4800,B9600);
  ParityType   = (EvenParity,OddParity,NoParity);
  TermCFG      = RECORD
                   LocalAreaCode : String[3];
                   UseTouchtones : Boolean;
                   DialOneFirst  : Boolean;
                   BaudRate      : BPS;
                   BitsPerChar   : Integer;
                   Parity        : ParityType
                 END;
```

```
CONST
  Config : TermCFG =
              (LocalAreaCode : '716';
               UseTouchtones : True;
               DialOneFirst  : True;
               BaudRate      : B1200;
               BitsPerChar   : 7;
               Parity        : EVEN_PARITY);
```

The structured constant declaration for **Config** must come after the type definition for **TermCFG**, otherwise the compiler would not know what a **TermCFG** was. Note that Turbo Pascal allows multiple **CONST** sections, and will allow you to place a **CONST** section *after* a **TYPE** section. This would not be allowed under Standard Pascal. Also, note that there is no **BEGIN/END** bracketing in the declaration of **Config**. The parentheses serve to set off the list of field values from the rest of your source code.

Set Constants

Declaring a set constant is not very different from assigning a set value to a set variable:

```
CONST
  Uppercase : SET OF Char = ['A'..'Z'];
```

The major difference is the notation used to represent nonprintable characters in a set of **Char**. Characters that do not have a printable symbol symbol associated with them may ordinarily be represented in a set constructor by the **Chr** transfer function:

```
MySet := [Chr(7),Chr(10),Chr(13)];
```

The **Chr** notation above will *not* work when declaring set constants. You have two alternatives:

1) Express the character as a control character by placing a caret symbol (^) in front of the appropriate character. The bell character, **Chr(7)**, would be expressed as ^G.

2) Express the character as its ordinal value preceded by a pound sign (#). The bell character would be expressed as #7. This notation is more useful for expressing characters falling in the high 128 bytes of type **Char**, corresponding to the line-drawing and foreign language characters on the IBM PC and many compatibles.

For example, the set of whitespace characters is a useful set constant:

```
CONST
  Whitespace : SET OF Char = [#8,#10,#12,#13,' '];
```

or, alternatively:

```
CONST
  Whitespace : SET OF Char = [^H,^J,^L,^M,' '];
```

The following three routines show you how to use set constants in simple character-manipulation tools. If you do a *lot* of character manipulation and find a great deal of use for these three set constants, you might also declare them at the global program level so that any part of the program can use them. Remember that, declared as they are here locally to their individual functions, they *cannot* be accessed from outside the function:

```
FUNCTION CapsLock(Ch : Char) : Char;

CONST
  Lowercase : SET OF Char = ['a'..'z'];

BEGIN
  IF Ch IN Lowercase THEN CapsLock := Chr(Ord(Ch)-32)
    ELSE CapsLock := Ch
END;


FUNCTION DownCase(Ch : Char) : Char;

CONST
  Uppercase : SET OF Char = ['A'..'Z'];

BEGIN
  IF Ch IN Uppercase THEN DownCase := Chr(Ord(Ch)+32)
    ELSE DownCase := Ch
END;


FUNCTION IsWhite(Ch : Char) : Boolean;

CONST
  Whitespace : SET OF Char = [#8,#10,#12,#13,' '];

BEGIN
  IsWhite := Ch IN WhiteSpace
END;
```

10 Pointers

10.1: STATIC AND DYNAMIC VARIABLES

Up to this point, we've made the assumption that all data items used in a Pascal program are known to the compiler when it compiles the program. That is, in order to use three integer variables **I**, **J**, and **K**, those three variables must be declared in the program source code:

```
VAR
  I,J,K : Integer;
```

The compiler uses this line to create three integer variables that remain available as long as the program is running. The three variables are given their proper places in the program's data area, and there they remain. Assuming it has only these three variables, the program cannot decide on the basis of its work that it needs another variable and then create one. What it got at compile time is what it has, period.

Variables of this sort are called "static variables." Pascal allows another kind of variable, which the program can, in fact, create as it needs them. These are called "dynamic variables."

Dynamic variables are identical in form to static variables. What is different about them is how they are created and how they are accessed. To understand that we must understand a new kind of data item: *the pointer.*

Almost by definition, the static data area of a program is set at compile time and can never change while the program is running. Dynamic variables, if they can be created and destroyed during a program's run, must exist somewhere else.

This "somewhere else" is a region of computer memory set aside by the compiler specifically to hold dynamic variables. It is called "heapspace" or "the heap." The word "heap" was coined in contrast to the word "stack," which is a memory-management method in which items are stored in strict order and must be stored and retrieved in that order. Like plates placed in a stack, the top one must come off first, then the next highest, and so on. In a heap of plates, the plates are in no special order. If you want a plate, you can reach in and take any of them you like in any order.

This is how it is with dynamic variables placed in heapspace. They are in no order, and they can be created and fetched at random.

10.2: DEFINING POINTER TYPES

The major purpose of pointer variables is to act as the bridge between the static data area and the heap. You must keep one thing in mind: *Dynamic variables have no names!* A pointer variable is, in one sense, a way of naming a dynamic variable; the only way, in fact, of "getting at" a dynamic variable. Pointer types are defined by the programmer just as enumerated types and structured types are:

```
TYPE
  AType = String[20];

  APtr  = ^AType;
```

With one rather arcane exception (the generic pointer, which I'll describe later), a pointer type is *always* defined as a pointer to some other type. The notation **^AType** is read, "pointer to **AType**." Pointers may point to any type except a file type.

One peculiarity of defining pointer types is that a pointer may be defined as pointing to a type which itself has not yet been defined. Consider the problem of defining a pointer variable that points to a record type containing (as a field) a pointer like itself, which points to a record of that type:

```
TYPE
  RecPtr     = ^DynaRec;
  DynaRec    = RECORD
                 DataPart : String;
                 Next     : RecPtr
               END;
```

Ordinarily, you cannot use an identifier until the compiler already knows what it is. However, there is a "chicken-and-egg" conflict here: If we define **RecPtr** first, the compiler doesn't know what **DynaRec** means. If we define **DynaRec** first, the compiler doesn't know what **RecPtr** means. So, by convention, Pascal allows us to define a pointer to a type which will later be defined. Of course, if you never get around to defining **DynaRec**, an error will be generated and the compilation will fail, with the message

```
Error 19: Undefined type in pointer definition (DynaRec)
```

Note that **DynaRec** must be defined somewhere within the *same* **TYPE** section as **^DynaRec**. Turbo Pascal allows multiple **TYPE** sections within a single program, but a pointer type and its base type must both be defined within the same section.

Records like **DynaRec** are central to the notion of "linked lists," which we will discuss in Section 21.2.

10.3: CREATING DYNAMIC VARIABLES

When a pointer variable is allocated by the compiler, its value is undefined, as with any static variable. It is even incorrect to say that it points to nothing. There is a way to explicitly force a pointer to point to nothing. An undefined pointer is simply undefined and should not be used until it has been given something (or nothing—see below) to point to.

Dynamic variables are created by a predefined Pascal procedure called **NEW**:

```
VAR
  APtr : RecPtr;

NEW(APtr);
```

The type **RecPtr** (as we saw above) was previously defined as a pointer to type **DynaRec**. Because **APtr** is of type **RecPtr**, the program (*not* the compiler; this is done at runtime!) knows to create a variable of type **DynaRec** off in heapspace. It then sets up **APtr** to point to the new dynamic variable of type **DynaRec**. **APtr**, which was previously undefined, now, indeed, points to something: to a brand new **DynaRec**.

Figure 10.1 shows a generalized diagram of this way of using **NEW**. Note in the diagram that **ATYPE** is not the *name* of the new dynamic variable, but its *type*.

Accessing a dynamic variable is done through its pointer. Going back to the text example, if we wish to assign some value to the **DataPart** field of the new **DynaRec**, we must use this notation:

```
APtr^.DataPart := 'Be dynamic!';
```

Assigning a dynamic variable (or a field of a dynamic record) to a static variable is done just as easily:

```
AString := APtr^.DataPart;
```

Accessing a dynamic variable through its pointer in this way is called "dereferencing" the pointer. I don't much care for the term because it seems to carry a certain sense of turning the pointer away from its referent, which of course isn't true.

There is no "clean" way to pronounce **APtr^** other than "that which pointer **APtr** points to" or (better) "**APtr**'s referent." The notation **APtr^** is the closest the new dynamic variable will ever come to having an actual name.

10.4: OPERATIONS ON POINTERS

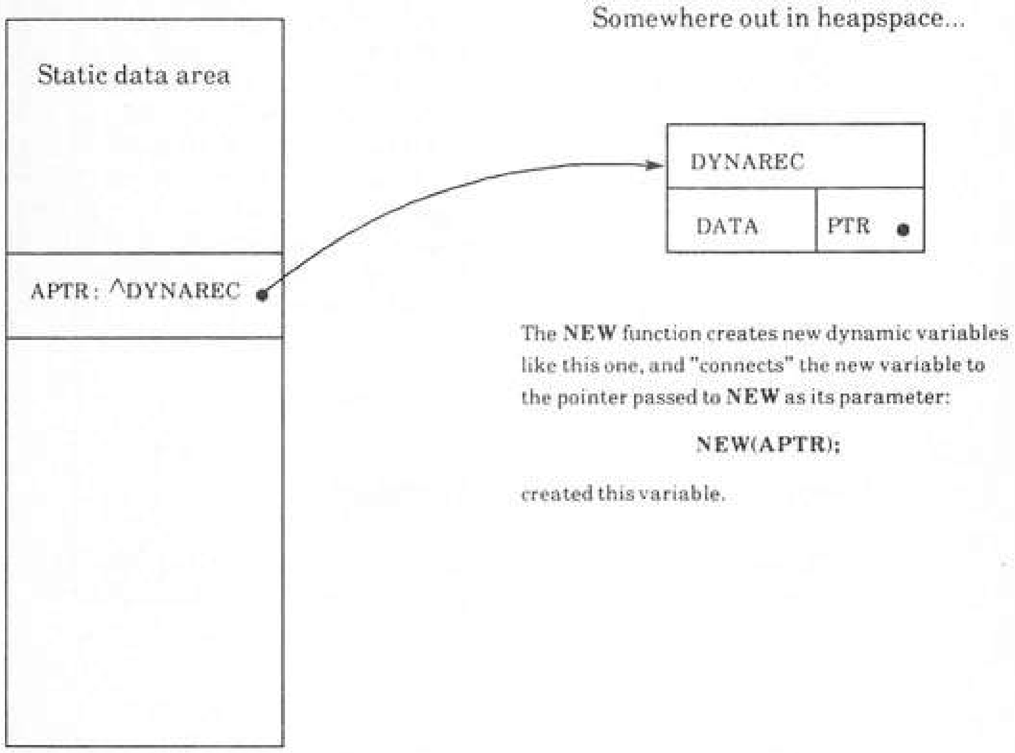
A pointer either is undefined, points to a dynamic variable, or points to nothing. Making a pointer point to nothing involves assigning it a special predefined value called **NIL**:

```
APtr := NIL;
```

It can now be said that **APtr** points to nothing. It is a good idea to assign **NIL** to all pointers soon after a program begins running. Using a pointer with a value of **NIL** is safer than using a pointer that is undefined.

Figure 10.1

A Pointer


Somewhere out in heapspace...
Static data area
APTR: ^DYNAREC
DYNAREC
DATA
PTR
The NEW function creates new dynamic variables like this one, and "connects" the new variable to the pointer passed to NEW as its parameter:
NEW(APTR);
created this variable.

Pointers may be compared for equality. When two pointers are said to be equal, they both point to the identical dynamic variable or they both have a value of **NIL**. Both pointers must be of the same type to be compared.

Comparing two undefined pointers (or one undefined pointer and one defined pointer) will return a meaningless value. *Don't do it.*

Pointers may be assigned to one another. Again, the two pointers must be of the same type, or the compiler will flag a type conflict error. The logic with pointer assignment is that, when you assign one pointer to a second pointer, the second pointer now points to the same dynamic variable to which the first pointer was pointing:

```
VAR
  APtr,BPtr : RecPtr;

NEW(APtr);          { APtr^ is a dynamic record        }
NEW(BPtr);          { BPtr^ is another dynamic record }
BPtr := APtr;       { What used to be BPtr^ is lost!   }
```

In this example, two pointers are each set to point to a new dynamic variable. You must keep in mind that each call to **NEW** produces a separate and distinct dynamic variable in heapspace. Then, however, the value of **APtr** is assigned in the assignment statement **BPtr**. Both pointers now point to the same dynamic record, which is the record **APtr** pointed to originally. **APtr^** and **BPtr^** are now identical records. Furthermore, the record that was created by the call **NEW(BPtr)** is now lost forever. *Once you "break" a pointer away from its dynamic variable, that variable is utterly inaccessible.*

This situation is worse, in fact, than it sounds. Even though the original **BPtr^** is inaccessible, *it still exists* and occupies memory that now cannot be used for anything else. Do *not* just cut a dynamic variable loose—unlike a kite or balloon which will fly away and cease being a problem, a "lost" dynamic variable will remain in existence for as long as the program that created it continues running. If you cut loose too many dynamic variables, you could conceivably run out of heapspace and be unable to create any new dynamic variables.

10.5: DISPOSING OF DYNAMIC VARIABLES

The way out is to clean house correctly when dynamic variables are no longer needed. Turbo Pascal contains a predefined procedure called **Dispose**, which is the opposite of NEW:

```
Dispose(APtr);
```

The **Dispose** procedure takes a pointer as its parameter. **Dispose** releases the memory occupied by **APtr^** (the dynamic variable pointed to by **APtr**) and causes the value of **APtr** to become undefined. Don't make the mistake of thinking that **Dispose** forces the value of its pointer parameter to **NIL**. The value becomes undefined, just as it was before you assigned a value to the pointer when the program began running.

The memory once occupied by **APtr^** goes back into a pool of free memory and becomes available for **NEW** to allocate to other dynamic variables.

10.6: GENERIC POINTERS AND POINTER CONSTANTS

Turbo Pascal allows a special type of pointer definition: the "generic," or untyped, pointer. It differs from the standard pointer in that it doesn't have a base type; it isn't defined as a pointer to some type or another, but simply as a variable of type **Pointer**:

```
VideoBuffer : Pointer;
```

Here, **VideoBuffer** is a pointer, but it cannot be directly dereferenced because it has no base type. In other words, you cannot use this syntax:

```
Char1 := VideoBuffer^;          { Will not compile! }
```

This statement is not only illegal (it generates **Error 26: Type conflict**), but it is also logically meaningless. If **VideoBuffer** has no base type, what sort of thing is **Video-Buffer^**?

What *can* be done (and we're getting a little ahead of ourselves here) is typecast the generic pointer's referent when assigning the referent to some variable:

```
Char1 := Char(VideoBuffer^);  { Will compile }
```

This becomes meaningful, because the compiler can take the first byte it finds at the location pointed to by **VideoBuffer** and consider that byte a value of type **Char**. Of course, you have to make sure that the byte that you're casting onto a character variable makes sense as a character. This is less of a problem with type **Char** (which legally can contain any 8-bit value) than with another type like **Boolean** that can have only two specific values, 1 and 0. If you cast a value of 17 onto a **Boolean**, the result is mostly confusion.

I haven't covered typecasting yet, but if you're interested in learning more about it, refer to Section 12.3.

Generic pointers are part of Turbo Pascal's suite of low-level system hooks and are actually a way of dressing up a raw machine address in a suit of clothes that Pascal will tolerate. Type **Pointer** is assignment-compatible with any other pointer type, including other generic pointers.

The uses for generic pointers generally involve situations in which the *size* of the referent is not known at compile time or may change even at run time. Video buffers are a good example of this: The EGA display board has a 4000-byte video buffer when operating in 25 × 80 mode, but a 6880-byte buffer when operating in 43 × 80 mode. If the user is allowed to switch modes while a program is operating, a pointer to a video buffer must be capable of dealing with two different sizes of buffers, and the sanest way to do that is to not entrust the pointer with any knowledge of the buffer's size at all. Other, even more arcane, tricks with generic pointers will come up in later sections.

Pointer types may be typed constants, but the only value a pointer constant may be initially assigned is **NIL**:

```
CONST
  IntPtr : ^Integer = Nil;
```

There's not a whole lot more to pointer variables themselves. Pointers are actually the glue that can hold together elaborate data structures formed from dynamic variables. There's a fair amount of Pascal to be learned before you should take on dynamic data structures like linked lists. Later, in Section 21.2, we'll look at them in greater detail.

Physically, a pointer is a machine address of some location in memory; specifically, of the first byte of the dynamic variable the pointer points to. Pointers in the PC's 8086 environment are 32-bit segments : offset machine addresses. A **NIL** pointer has all 4 bytes set to binary 0.

Unless your application absolutely requires it, you should avoid tinkering with pointers beneath the level of the language itself. If you do, you lose any protection against stray memory references that Turbo Pascal's runtime library provides. After all, a pointer is just a memory reference. If you accidentally create a pointer that points to an area somewhere in the middle of your operating system and then store a dynamic variable at that location, you will crash resoundingly.

10.7: GETMEM AND FREEMEM

When you create a dynamic variable with **New**, you don't have to tell Turbo Pascal how much memory to allocate on the heap for that variable. Apart from generic pointers, a pointer is always a pointer to something, and that something, a data type, has a size that is fixed at compile time and never changes. Therefore, Turbo Pascal can tell from the type to which the pointer points how much memory will be needed for the dynamic variable.

Dynamic variables, **New**, and **Dispose** are all part of Standard Pascal. They allow you to create variables on the heap without specifying how many or in what order they are created. Turbo Pascal lets you take it one step further: You can allocate blocks of memory on the heap for a unit of data whose size is *not* known at compile time.

Just as Turbo Pascal extends the Standard Pascal pointer concept with generic pointers, it provides generic versions of **New** and **Dispose**. These are **GetMem** and **FreeMem**. **GetMem** is the generic counterpart of **New**. It allocates a block of memory on the heap of a specified size:

```
PROCEDURE GetMem(VAR AnyPtr : <any pointer type>; Size : Word);
```

The pointer variable passed as the first parameter may be any type of pointer at all, including a generic pointer. Furthermore, if the pointer is a typed pointer, the size of the type to which it points need not be the same as the value passed in **Size**. You should remember, however, that it isn't an especially good idea to use typed pointers with **GetMem** if you can avoid it.

On returning from a call to **GetMem**, **AnyPtr** will point to a block of memory on the heap consisting of **Size** bytes. As with **New**, if Turbo Pascal's heap manager can't find a single block of heap memory large enough to satisfy the request, a runtime error will be generated. This situation can be improved somewhat by installing a custom heap error handler (see Section 21.5).

Just as **GetMem** is the generic counterpart of **New**, **FreeMem** is the generic counterpart of **Dispose**:

```
PROCEDURE FreeMem(VAR AnyPtr : <any pointer type>; Size : Word);
```

FreeMem deallocates the memory allocated to the referent of **AnyPtr** and forces the value of **AnyPtr** to be undefined (*not* **NIL**!). It is crucial that the value passed in **Size** is *precisely* the same value that had previously been allocated to the referent of **AnyPtr**. If not, you will corrupt the heap, and unexpected (and certainly unwanted) things may begin to happen. The burden is on you to make sure that this does not happen; there is nothing in Turbo Pascal to protect the heap from careless use of **FreeMem**.

GetMem and **FreeMem** are most often used to create on the heap buffers whose size is not known at compile time. The best example of this involves the saving of portions of a graphics screen on the heap by using the **GetImage** procedure from the **Graph** unit. (**Graph** will be discussed in detail in Chapter 22.) Saving a small portion of the screen requires a certain amount of heap memory, and saving a larger portion requires more. Using **GetMem** permits you to allocate only as much heap space as you need to save a piece of the graphics screen: no more, no less. The **Move** procedure (see Section 23.4) is another way to move data into and out of a buffer allocated with GetMem.

Good examples of **GetMem** and **FreeMem** in use may be found in the **Scribble** program and the **PullDown** unit it uses. These are described in Chapter 22, in connection with the Borland Graphics Interface (BGI).

Taken together, typecasting, generic pointers, the **Move** procedure, the @ function, **GetMem**, **FreeMem**, and the **Ptr** function provide a nearly complete end-run around Standard Pascal's strong typing straitjacket. Turbo Pascal is now the functional equivalent of the C language, with all the power—and danger—implied therein. There is a whole second level of things you can do with pointers that has less to do with the heap than with manipulating the system at a low level. This involves reading values from DOS and BIOS work areas, and working directly with video bitmaps (see Chapter 22), among other things. You should be fairly comfortable with the standard features of Turbo Pascal and at least passingly comfortable with machine internals before attempting things of this sort.

11 Operators and Expressions

Up to this point, we have covered in depth the various data types that Pascal offers. Data are representations of concepts that we want a program to manipulate. From here on, we will be speaking largely of how the Pascal language manipulates its data.

As previously explained, a variable is a container in memory laid out by the compiler. It has a particular size and shape defined by its type. Into a variable you load an item of data that conforms to the size and shape of the variable. This item of data is called a "value."

You are probably familiar with this symbol, perhaps the most fundamental operator in Pascal: :=. It is the assignment operator. The assignment operator is usually used to place values into variables. We've been using it informally all along and will discuss it in some detail in Section 12.1. In case you aren't sure by now how it works, consider this simple assignment statement:

```
I := 17;
```

A value on the right side of the assignment operator is assigned to the variable on the left side of the assignment operator. In an assignment statement, there is always a variable on the left side of the assignment operator. On the right side may be a constant, a variable, or an expression.

An expression in Pascal is a combination of data items and operators which eventually "cooks down" to a single value. Data items are constants and variables. Operators are special symbols that perform some action involving the value or values given to it. These values are called an operator's "operands."

The simplest and most familiar examples of expressions come to us from arithmetic. This is an expression:

```
17 + 3
```

The addition operator performs an add operation on its two operands, 17 and 3. The value of the expression is 20.

An expression like 17 + 3, while valid, would not be used in a real program, in which the value "20" would suffice. Considerably more useful are expressions which involve variables. For example:

```
3.14159 * Radius * Radius
```

This expression's value is recognizable as the area of the circle defined by whatever value is contained in **Radius**.

ISO Standard Pascal includes a good many different operators for building expressions, and Turbo Pascal enhances ISO Pascal with a few additional operators. They fall into a number of related groups depending on what sort of result they return: relational, arithmetic, set, and logical (also called "bitwise") operators.

11.1: RELATIONAL OPERATORS

The relational operators are used to build Boolean expressions, that is, expressions that evaluate to a Boolean value of **True** or **False**. Boolean expressions are the most widely used of all expressions in Pascal. All the looping and branching statements in Pascal depend on Boolean expressions.

A relational operator causes the compiler to compare its two operands for some sort of relationship. The Boolean value that results is calculated according to a set of well-defined rules as to how data items of various sorts relate to one another.

Table 11.1 summarizes the relational operators implemented in Turbo Pascal. All return Boolean results.

You will recall that scalar types are types with a limited number of ordered values. **Char**, **Boolean**, **Integer**, **Byte**, and enumerated types are all scalars. Real numbers, strings, structured types (records and arrays), and sets are not scalars.

The three relational operators that involve sets, such as set membership and the two set-inclusion operators, will be discussed along with the operators that return set values in Section 11.3. You should note that the set inclusion operators share symbols with the greater than or equal to/less than or equal to operators, but the sense of these two types of operations are radically different.

Table 11.1
Relational operators implemented in Turbo Pascal.

Operator	*Symbol*	*Operand Type*	*Precedence*
Equality	=	Scalar, set, string, pointer, record	5
Inequality	<>	Scalar, set, string, pointer, record	5
Less than	<	Scalar, string	5
Greater than	>	Scalar, string	5
Less than or equal	<=	Scalar, string	5
Greater than or equal	>=	Scalar, string	5
Set membership	IN	Set, set members	5
Set inclusion, left in right	<=	Set	5
Set inclusion, right in left	>=	Set	5
Negation	NOT	Boolean	2
Conjunction	AND	Boolean	3
Disjunction	OR	Boolean	4
Exclusive OR	XOR	Boolean	4

Equality

If two values compared for equality are the same, the expression will evaluate as **True**. In general, for two values to be considered equal by Turbo Pascal's runtime code, they must be identical on a bit-for-bit basis. This is true for comparisons between like types. Most comparisons must be between values of the same type.

The exceptions are comparisons between numeric values expressed as different types. Turbo Pascal allows comparisons rather freely among integer types and real number types, but this sort of type-crossing must be done with great care. In particular, do not compare calculated reals (real number results of a real number arithmetic operation) for equality, either with other reals or with numeric values of other types. Rounding effects may cause real numbers to appear to be unequal to compiled code, even though the mathematical sense of the calculation would seem to make them equal.

Integer types **Byte**, **ShortInt**, **Word**, **Integer**, and **LongInt** may be freely compared among themselves.

Two sets are considered equal if they contain exactly the same members. The two sets must, of course, be of the same base type to be compared at all. Two pointers are considered equal if they both point to the same dynamic variable. Two pointers are also considered equal if they both point to **NIL**.

Two records are considered equal if they are of the same type (you cannot compare records of different types) and if each field in one record is identical bit-for-bit to its corresponding field in the other record. Remember that you *cannot* compare records, even of the same type, using the greater than/less than operators (>, <, >=, or <=).

Two strings are considered equal if they both have the same logical length (see Section 9.5) and contain the same characters. This makes them identical bit-for-bit out to the logical length, which is the touchstone for all like-type equality comparisons under Turbo Pascal. Remember that this makes leading and trailing blanks significant:

```
'    Eriador' <> 'Eriador'     { True! }
'Eriador    ' <> 'Eriador'     { True! }
```

Inequality

The rules for testing for inequality are exactly the same as the rules for equality. The only difference is that the Boolean state of the result is reversed:

```
17 = 17                 { True  }
17 <> 17                { False }
42 = 17                 { False }
42 <> 16                { True  }
```

In general, you can use the inequality operator anywhere you can use the equality operator.

Pointers are considered unequal when they point to different dynamic variables, or when one points to **NIL** and the other does not. The bit-for-bit rule is again applied: Even a 1-bit difference found during a type comparison means the two compared operands are unequal. The warning applied to rounding errors produced in calculated reals applies to inequality comparisons as well.

Greater Than/Less Than

The four operators greater than, less than, greater than or equal to, and less than or equal to, add a new dimension to the notion of comparison. They assume that their operands always exist in some well-defined order by which the comparison can be made.

This immediately disqualifies pointers, sets, and records. Given the way pointers are defined, saying one pointer is greater than another simply makes no sense. You could argue that, because pointers are physical addresses, one pointer will always be greater than or less than another, nonequal pointer. This may be true. In the spirit of the Pascal language, however, details about how pointers are implemented are hidden from the programmer at the level of Pascal.

The same applies to sets and records. Ordering them makes no logical sense; thus, operators involving an implied order cannot be used with them.

With scalar types, a definite order is part of the type definition. For integer types **ShortInt**, **Byte**, **Integer**, **Word**, and **LongInt**, the order is obvious from our experience with arithmetic; integers model whole numbers within specific bounds.

The **Char** and **Byte** types are both limited to 256 possible values, and both have an order implied by the sequence of binary numbers from 0 to 255. The **Char** type is ordered (at least for the lower 128 values) by the ASCII character set, which makes the following expressions evaluate to **True**:

```
'A' < 'B'
'a' > 'A'
'@' < '['
```

The higher 128 values assignable to **Char** variables have no standard characters, but they still exist in fixed order and are numbered from 128 to 255.

Enumerated types are limited to no more than 255 different values, and usually have fewer than 10 or 12. Their fixed order is the order in which the values were given in the definition of the enumerated type:

```
TYPE
  Colors = (Red,Orange,Yellow,Green,Blue,Indigo,Violet);
```

This order makes the following expressions evaluate to **True**:

```
Red < Green
Blue > Yellow
Indigo < Violet
```

The ordering of string values involves two components: the length of the string and the ASCII values of the characters present in the string. Essentially, Turbo Pascal begins by comparing the first characters in the two strings being compared. If those two characters are different, the test stops there, and the ordering of the two strings is based on the relation of those two first characters. If the two characters are the same, the second characters in each string are compared. If they turn out to be the same, then the third characters in both strings are compared.

This process continues until the code finds two characters that differ, or until one string runs out of characters. In that case, the longer of the two is considered to be greater than the shorter. All of the following expressions evaluate to True:

```
'AAAAA' > 'AAA'
'B' > 'AAAAAAAAAAAAA'
'AAAAB' > 'AAAAAAAAAA'
```

NOT, AND, OR, and XOR

There are four operators that operate on Boolean operands: **NOT**, **AND**, **OR** and **XOR**. These operators are sometimes set apart as a separate group called "Boolean operators." In some ways, they have more in common with the arithmetic operators than with the relational operators. They do not test a relationship that already exists between two operands. Rather, they combine their operands according to the rules of Boolean algebra to produce a new value that becomes the value of the expression.

The simplest of the four is **NOT**, which takes only one Boolean operand. The operand must be placed after the **NOT** reserved word. **NOT** negates the Boolean value of its operand:

```
NOT False         { Expression is True    }
NOT True          { Expression is False   }
```

Some slightly less simplistic examples:

```
NOT (6 > I)      { True for I < 6    }
NOT (J = K)      { True for J <> K   }
```

The parentheses indicate that the expression within the parentheses is evaluated first, and only then is the resultant value acted upon by **NOT**. This expression is an instance in which order of evaluation becomes important. We will discuss this in detail in Section 11.5.

AND (also known as "conjunction") requires two operands, and follows this rule: *If both operands are **True**, the expression returns **True**; otherwise, the expression returns **False**.* If either operand or both operands have the value **False**, the value of the expression as a whole will be **False**.

Some examples:

```
True AND True                    { Expression is True  }
True AND False                   { Expression is False }
False AND True                   { Expression is False }
False AND False                  { Expression is False }

(7 > 4) AND (5 <> 3)             { Expression is True  }
(16 = (4 * 4)) AND (2 <> 2)      { Expression is False }
```

All these example expressions use constants, and thus are not realistic uses of **AND** within a program. We present them this way so the logic of the statement is obvious without having to remember what value is currently in a variable present in an expression. We'll be presenting some real-life coding examples of the use of **NOT**, **AND**, and **OR** in connection with the discussion of order of evaluation in Section 11.5.

OR (also known as "disjunction") also requires two operands, and follows this rule: *If either or both operands are **True**, the expression returns **True**; only if both operands are **False** will the expression return **False**.*

Some examples, again using constants:

```
True OR True                { Expression is True  }
True OR False               { Expression is True  }
False OR True               { Expression is True  }
False OR False              { Expression is False }

(7 > 4) OR (5 = 3)          { Expression is True  }
(2 < 1) OR (6 <> 6)         { Expression is False }
```

Finally, **XOR**, which also requires two operands, follows this rule: *If both operands have the same Boolean value, **XOR** returns **False**; only if the operands have unlike Boolean values will **XOR** return **True**.*

Some examples:

```
True XOR True               { Expression is False }
True XOR False              { Expression is True  }
False XOR True              { Expression is True  }
False XOR False             { Expression is False }
```

Greater Than or Equal To/Less Than or Equal To

These two operators are both combinations of two operators. These combinations are so convenient and so frequently used that they were welded together to form two single operators with unique symbols: >= (read "greater than or equal to") and <= (read "less than or equal to").

When you wish to say:

```
X >= Y
```

you are in fact saying:

```
(X > Y) OR (X = Y)
```

and when you wish to say:

```
X <= Y
```

you are in fact saying:

```
(X < Y) OR (X = Y)
```

The rules for applying >= and <= are exactly the same as those for applying > and <. They may take only scalars or strings as operands.

11.2: ARITHMETIC OPERATORS

Manipulating numbers is done with arithmetic operators, which, along with numeric variables, form arithmetic expressions. About the only common arithmetic operator not found in Pascal is the exponentiation operator, that is, the raising of one number to a given power. We will build in a function 16.4 that raises one number to the power of another. Table 11.2 summarizes the arithmetic operators implemented in Turbo Pascal.

Table 11.2
Arithmetic operators implemented in Turbo Pascal.

Operator	*Symbol*	*Operand Types*	*Result Type*	*Precedence*
Addition	+	Integer, real, byte	Same	4
Sign inversion	−	Integer, real	Same	1
Subtraction	−	Integer, real, byte	Same	4
Multiplication	*	Integer, real, byte	Same	3
Integer division	DIV	Integer, byte	Same	3
Real division	/	Integer, real, byte	Real	3
Modulus	MOD	Integer, byte	Same	3

The operands column lists those data types which a given operator may take as operands. The compiler is fairly free about allowing you to mix types of numeric variables within an expression. In other words, you may multiply bytes by integers, add reals to bytes, multiply integers by bytes, and so on. For example, these expressions are all legal in Turbo Pascal:

```
VAR
  I,J,K  : Integer;
  R,S,T  : Real;
  A,B,C  : Byte;
  U,V    : Single;
  W,X    : Double;
  Q      : ShortInt;
  L      : LongInt;

I * B          { Integer multiplied by byte }
R + J          { Integer added to real      }
L * Q          { LongInt multiplied by ShortInt }
C + (R * I)    { Etc. }
J * (A / S)
```

The result type column in the table indicates the data type which the value of an expression incorporating that operator may take on. Ordinarily, Pascal is very picky about the assignment of different types to one another in assigment statements. This "strict type checking" is relaxed to some extent in simple arithmetic expressions. In fact, numeric types may be mixed fairly freely within expressions as long as a few rules are obeyed:

1. Any expression including a floating point value may be assigned only to a floating point variable.
2. Expressions containing floating point division / may be assigned only to a floating point variable *even if all the operands are integer types.*

Failure to follow these rules will generate:

```
Error 26: Type mismatch.
```

Outside of the two mentioned restrictions, however, a numeric expression may be assigned to any numeric variable, assuming the variable has sufficient range to contain the value of the expression. For example, if an expression evaluates to 14,000, you should not assign the expression to a variable of type **Byte**, which can contain only values from 0 to 255. Program behavior in such a case is unpredictable.

Addition, subtraction, and multiplication are handled the same way arithmetic ordinarily is handled with pencil or calculator, and we won't need to describe these methods here.

The Strange Case of Comp

Newcomers to Pascal will probably have some trouble deciding exactly what type **Comp** really is. Under Turbo Pascal 4.0, **Comp** is only available on machines containing a math coprocessor like the 8087 or 80287. Using release 5.0, **Comp** may be used in machines without math coprocessors by taking advantage of 5.0's floating point emulation feature. **Comp** is related to the defunct **BCD** numeric type that died with Version 3.0 Turbo BCD Pascal. Although nominally an integer in function (it takes no decimal part), it acts much as a floating point type in most circumstances. For example:

- **Comp** may be assigned a value from an expression containing floating point values. Any decimal parts are rounded to the closest integer.
- **Comp** *must* be used with floating point division (the / operator) rather than integer division (the **MOD** and **DIV** operators.) Trying to mix **Comp** with **MOD** and **DIV** will generate

```
Error 41: Operand types do not match operator.
```

- Unless you specify a numeric format, **Comp** values will be displayed in exponential format, just as with floating point values.

Something to keep in mind: ***Comp** is not a scalar type.* You cannot use the **Pred** or **Succ** functions on **Comp** variables, and **Comp** variables cannot index arrays. In a sense, **Comp** is a floating point type that has been designed to give it absolute whole-number precision throughout its range. This makes it suitable for numeric computations, and it will probably be most valuable in financial applications, in which **Comp** values may represent monetary amounts in either cents or mills. (A mill is a thousandth of a dollar, although the concept is rarely used outside the securities industry.) It is much more suitable than type **LongInt** for financial values, because **Comp** is accurate to 17 significant figures and **LongInt** only to a measly 10 significant figures, or roughly two billion. See Section 8.7 for more on **Comp**.

Sign Inversion

Sign inversion is a "unary" operator; that is, it takes only a single operand. What it does is reverse the sign of its operand. It will make a positive quantity negative or a negative quantity positive. It does not affect the absolute value (the distance from zero) of the operand.

Note that sign inversion cannot be used with **Byte** or **Word**. These two types are "unsigned," that is, they are never considered negative.

Division

There are three distinct division operators in Pascal. One supports floating point division, and the other two support division for integer types. Floating point division / may take operands of any numeric type, but it always produces a floating point value, complete with decimal part. Attempting to assign a floating point division expression to an integer type will generate error #26, *even if all numeric variables involved are integers:*

```
VAR
  I,J,K : Integer;

I := J / K;           { Won't compile!!! }
```

Division for numbers that cannot hold decimal parts is much the same as division as it is first taught to grade-schoolers: When one number is divided by another, two numbers result. One is a whole number quotient; the other is a whole number remainder.

In Pascal, integer division is actually two separate operations that do not depend on one another. One operator, **DIV**, produces the quotient of its operands:

```
J := 17;
K := 3;
I := J DIV K;         { I is assigned the value 5  }
```

No remainder is generated at all by **DIV**, and the operation should not be considered incomplete. If you wish to compute the remainder, the modulus operator (**MOD**) is used:

```
I := J MOD K;       { I is assigned the value 2  }
```

Assuming the same values given above for **J** and **K**, the remainder of dividing **J** by **K** is 2. The quotient is not calculated at all or calculated internally and thrown away; only the remainder is returned.

11.3: SET OPERATORS

Set operators are used to manipulate values of type **SET**. To a great extent, they follow the rules of set arithmetic you may have learned in grade school. Table 11.3 summarizes the set operators implemented in Turbo Pascal. All of them take set operands and return set values.

Table 11.3
Set operators implemented in Turbo Pascal.

Operator	*Symbol*	*Precedence*
Set union	+	4
Set difference	–	4
Set intersection	*	4

Set Union

The *union* of two sets is the set that contains as members all members contained in both sets. The symbol for the set union operator is the plus sign +, just as for arithmetic addition. An example:

```
VAR
  SetX, SetY, SetZ : SET OF Char;

SetX := ['Y','y','M','m'];
SetY := ['N','n','M','m'];
SetZ := SetX + SetY;
```

SetZ now contains **'Y'**, **'y'**, **'N'**, **'n'**, **'M'**, and **'m'**. Note that, although **'M'** and **'m'** exist in both sets, each appears but once in the union of the two sets. A member is merely present or not present in the set; it is meaningless to speak of how many times a member is present in a set. By definition, each member is present only once or not present at all.

Set Difference

The *difference* between two sets is conceptually related to subtraction. Given two sets **SetA** and **SetB**, the difference between them, expressed as **SetA - SetB**, is the set consisting of the elements of **SetA** that remain once all the elements of **SetB** have been removed from it. For example:

```
SetX := ['Y','y','M','m'];
SetY := ['N','n','M','m'];
SetZ := SetX * SetY;
```

SetZ now contains **'Y'** and **'y'**.

Set Intersection

The intersection of two sets is the set which contains as members only those members contained in *both* sets. The symbol for set intersection is the asterisk *, just as for arithmetic multiplication. For example:

```
SetA := ['Y','y','M','m'];
SetB := ['N','n','M','m'];
SetX := SetA - SetB;
```

SetZ now contains **'M'** and **'m'**, which are the only two members contained in both sets.

The Set Relational Operators

The set operators just described work with set operands to produce new set values. We have briefly mentioned the relational operators that test relationships between sets and return Boolean values depending on those relationships.

Sets, for example, can be equal to one another if they both have the same base type and contain the same elements. Two sets that are not of the same base type will generate a type mismatch error at compile time if you try to compare them:

```
VAR
  SetX : SET OF Char;
  SetQ : SET OF Color;
  OK   : Boolean;

OK := SetX = SetQ;          { Will trigger error #26! }
```

This holds true for *all* set relational operators, not just equality.

The most important set relational operator is the inclusion operator **IN**. Operator **IN** tests whether a value of a set's base type is a member of that set:

```
VAR
  Ch : Char;

Read(Ch);                   { From the keyboard }
IF Ch IN ['Y','y'] THEN
  Write('Yes indeed!');
```

This example shows a clean and easy way to discern whether a user has typed the letter Y in response to a prompt. The **IN** operator tests whether the typed-in character is a member of the set constant [**'Y','y'**]. (**IN** works just as well with set variables.) The alternative would be to use a more complicated Boolean expression:

```
IF (Ch = 'Y') OR (Ch = 'y') THEN
  Write('Yes indeed!');
```

The **IN** operator is also faster than using a series, especially a long series, of relational expressions linked with the **OR** operator.

The greater than (>) and less than (<) operators make no sense when applied to sets, because sets have no implied order to their values. However, there are two additional set relational operators that make use of the same symbols as used by the greater than or equal to (>=) and less than or equal to (<=) operators. These are the set inclusion operators.

Inclusion of left in right (<=) tests whether all members of the set on the left are included in the set on the right. Inclusion of right in left (>=) tests whether all members of the set on the right are included in the set on the left. The actions these two operators take is identical except for the orientation of the two operands with respect to the operator symbol. Given two sets, **Set1** and **Set2**, the expression:

```
(Set1 <= Set2) = (Set2 >= Set1)
```

will always evaluate to **True**. These operators are very handy for testing and manipulating characters in a text stream. For example:

```
VAR
  Vowels,Alphabet,Samples : SET OF Char;

Vowels:=['A','E','I','O','U'];
Alphabet:=['A'..'Z'];             { Set of all UC letters }
Samples:=['A','D','I','Q','Z'];

IF Samples <= Vowels THEN Write('All samples are vowels.');
IF NOT(Samples <= Vowels) AND (Samples <= Alphabet) THEN
  Write('Some or all samples are uppercase letters.');
IF NOT(Alphabet >= Samples) THEN
  Write('Some samples are not uppercase letters.');
```

In addition to demonstrating the set inclusion operators, these examples also show some uses of the **AND** and **NOT** operators. Given the members assigned to **Samples** in the above example, this output will be displayed:

```
Some or all samples are uppercase letters.
```

Before going on, jot down two sets of characters that would trigger the other two messages in the above example.

11.4: BITWISE OPERATORS

Turbo Pascal allows you to read and write I/O ports and other low-level data, much of which are *bit-mapped*; that is, certain bits have certain meanings apart from all other bits and must be examined, set, and interpreted individually. The way to do this is through the *bitwise* logical operators. Associated with the bitwise logical operators are the shift operators, **SHL** and **SHR**. We will speak of these shortly.

We have previously dealt with the **AND**, **OR**, **XOR**, and **NOT** operators, which work on Boolean operands and return Boolean values. The bitwise logical operators are another variety of **NOT**, **AND**, **OR**, and **XOR**. They work with operands of all integer types (**Integer**, **Word**, **LongInt**, **ShortInt**, and **Byte**), and they apply a logical operation upon their operands 1 bit at a time.

Table 11.4 summarizes the bitwise logical operators and the shift operators:

The best way to approach all bitwise operators is to work in true binary notation, in which all numbers are expressed in base 2 and the only digits are 1 and 0. The bitwise operators work on one binary digit at a time. The result of the various operations on 0 and 1 values is best summarized by four "truth tables":

NOT	*AND*	*OR*	*XOR*
NOT 1 = 0	0 AND 0 = 0	0 OR 0 = 0	0 XOR 0 = 0
NOT 0 = 1	0 AND 1 = 0	0 OR 1 = 1	0 XOR 1 = 1
	1 AND 0 = 0	1 OR 0 = 1	1 XOR 0 = 1
	1 AND 1 = 1	1 OR 1 = 1	1 XOR 1 = 0

When you apply bitwise operators to two 8-bit or two 16-bit data items, it is the same as applying the operator between each corresponding bit of the two items. For example, the following expression evaluates to **True**:

```
$80 = ($83 AND $90)      { All in hexadecimal  }
```

Table 11.4
Bitwise logical operators and shift operators.

Operator	*Symbol*	*Operand Types*	*Result Type*	*Precedence*
Bitwise NOT	NOT	All integer types	Same	2
Bitwise AND	AND	All integer types	Same	3
Bitwise OR	OR	All integer types	Same	4
Bitwise XOR	XOR	All integer types	Same	4
Shift Right	SHR	All integer types	Same	3
Shift Left	SHL	All integer types	Same	3

Why? Think of the operation $83 & $90 this way:

```
Hex           Binary

$83 =   1 0 0 0 0 0 1 1
             AND
$90 =   1 0 0 1 0 0 0 0
               =
$80 =   1 0 0 0 0 0 0 0
```

Read *down* from the top of each column in the binary number, and compare the little equation to the truth table for bitwise **AND**. If you apply bitwise **AND** to each column, you will find the bit pattern for the number $80 to be the total result.

Now, what good is this? Suppose you want to examine only 4 of the 8 bits in a variable of type **Byte**. The bits are numbered 0 through 7 from the right. The bits you need are bits 2 through 5. The way to do it is to use bitwise **AND** and what we call a "mask":

```
VAR
  GoodBits, AllBits : Byte;

GoodBits := AllBits AND $3C;
```

To see how this works, let's again "spread it out" into a set of binary numbers:

```
AllBits      =  X X X X X X X X
                      AND
$3E (mask) =  0 0 1 1 1 1 0 0
                       =
GoodBits     =  0 0 X X X X 0 0
```

Here, X means "*either* 1 or 0." Again, follow the eight little operations down from the top of each column to the bottom. The 0 bits present in four of the eight columns of the mask, $3C, force those columns to evaluate to 0 in **GoodBits**, regardless of the state of the corresponding bits in **AllBits**. Go back to the truth table if this is not clear: *If either of the two bits in a bitwise AND expression is 0, the result will be 0.*

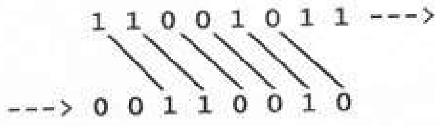
This way, we can assume that bits 0,1,7, and 8 in **GoodBits** will always be 0, and we can ignore them while we test the others.

We will see more examples of the use of the bitwise logical operators in Section 23.5.

Shift Operators

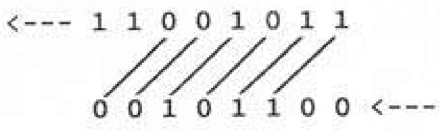
We've looked at bit patterns as stored in integer types and ways in which we can alter those patterns by logically combining bit patterns with bitmasks. Another way to alter bit patterns in integer types is with the shift operators, **SHR** and **SHL**. **SHR** stands for SHift Right, **SHL** for SHift Left.

Both operators are best understood by looking at a bit pattern before and after the operator acts upon it. Start with the value $CB (203 decimal) and shift it 2 bits to the right as the **SHR** operator would do:



The result byte is $32 (50 decimal). The 2 1-bits on the right end of the original $CB value are shifted off the end of the byte (into the "bit bucket," some say) and are lost. To take their place, 2 0-bits are fed into the byte on the left end.

SHL works identically, but in the other direction. Let's shift $CB to the left with **SHL** and see what we get:



Again, we lose two bits off the end of the original value, but this time they drop off the left end, and 2 0-bits are added in on the right end. What was $CB is now $2C (44 decimal).

Syntactically, **SHL** and **SHR** are like the arithmetic operators. They act with the number of bits to be shifted to form an expression, the resulting value of which you assign to another variable:

```
Result := Operand <SHL/SHR> <number of bits to shift>;
```

Some examples:

```
VAR
  B,C : Byte;
  I,J : Integer;

I := 17;
J := I SHL 3;     { J now contains 136 }
B := $FF;
C := B SHR 4;     { C now contains $0F }
```

It would be a good exercise to work out these two examples shifts on paper, expressing each value as a binary pattern of bits and then shifting them.

It is interesting to note that the shift operators are extremely fast means of multiplying and dividing a number by a power of 2. Shifting a number 1 bit to the left multiplies it by 2. Shifting it 2 bits to the left multiplies it by 4, and so on. In the example above, we shifted 17 by 3 bits, which multiplies it by 8. Sure enough, $17 \times 8 = 136$.

It works the other way as well. Shifting a number 1 bit to the right divides it by 2. Shifting 2 bits to the right divides it by 4, and so on. The only thing to watch is that there is no remainder on divide and nothing to notify you if you overflow on a multiply. It is a somewhat limited form of arithmetic, but in time-critical applications, you'll find it is *much* faster than the more generalized multiply and divide operators.

Examples of **SHL** and **SHR** in use will be found in Section 23.5.

11.5: ORDER OF EVALUATION

Mixing several operators within a single expression can lead to problems for the compiler. Eventually the expression must be evaluated down to a single value, but in what order are the various operators to be applied? Consider the ambiguity in this expression:

```
7 + 6 * 9
```

How will the compiler interpret this? Which operator is applied first? As you might expect, the authors of Turbo Pascal set down rules that dictate how expressions containing more than one operator are to be evaluated. These rules define *order of evaluation.*

To determine the order of evaluation of an expression, the compiler must consider three factors: precedence of operators, left-to-right evaluation, and parentheses.

Precedence

All operators in Turbo Pascal have a property called "precedence." Precedence is a sort of evaluation prioritizing system. If two operators have different precedences, the one with higher precedence is evaluated first. There are five degrees of precedence: 1 is the highest and 5 the lowest.

When we summarized the various operators in tables, the rightmost column contained each operator's precedence. The sign inversion operator has a precedence of 1. No other operator has a precedence of 1. Sign inversion operations are always performed before any other operations, assuming parentheses are not present (we'll get to that shortly). Logical and bitwise **NOT** operators have a precedence of 2. For example:

```
VAR
  OK,FileOpen : Boolean;

IF NOT OK AND FileOpen THEN CallOperator;
```

How is the expression, **NOT OK AND FileOpen** evaluated? **NOT OK** is evaluated first. The Boolean result is then **AND**ed with the Boolean value in **FileOpen** to yield the final Boolean result for the expression. If that value is **True**, the procedure **CallOperator** is executed.

Left to Right Evaluation

The previous example was clear cut, given that **NOT** has a higher precedence than **AND**. But, as you can see from the tables, many operators have the same precedence value. Addition, subtraction, and set intersection are only a few of the operators with a precedence of 4, and *all* relational operators have a precedence of 5.

When the compiler is evaluating an expression and it confronts a choice between two operators of equal precedence, it evaluates them in order from left to right. For example:

```
VAR
  I,J,K : Integer;

J := I * 17 DIV K;
```

The * and **DIV** operators both have a precedence of 3. To evaluate the expression **I * 17 DIV K**, the compiler must first evaluate **I * 17** to an integer value and then integer divide that value by **K**. * is to the left of **DIV**, and so it is evaluated before **DIV**.

You should note that left-to-right evaluation happens *only* when it is not clear from precedence (or parentheses, see below) which of two operators must be evaluated first.

Setting Order of Evaluation with Parentheses

There are situations in which the previous two rules break down. How would the compiler establish order of evaluation for this expression:

```
I > J AND K <= L
```

The idea here is to test the Boolean values of two relational expressions. The precedence of **AND** is greater than the precedence of any relational operator like <and >=. Thus, the compiler would attempt to evaluate the subexpression **J** AND **K** first.

Actually, this particular expression does not even compile; Turbo Pascal will flag an error **#1** as soon as it finishes compiling **J AND K** and sees another operator ahead of it.

The only way out of this problem is to use parentheses. In the rules of Pascal, as in the rules of algebra, parentheses override *all* other order-of-evaluation rules. To make the offending expression pass muster, you must rework it this way:

```
(I > J) AND (K <= L)
```

Now, the compiler first evaluates **(I>J)** to a Boolean value, then **K <=L)** to another Boolean value, then submits those two Boolean values as operands to **AND**. **AND** happily generates a final Boolean value for the entire expression.

This is one case, and a fairly common one, in which parentheses are required to compile the expression without errors. However, there are many occasions when parentheses will make an expression more readable, even though, strictly speaking, the parentheses are not required:

```
(PI * Radius) + 7
```

Here, not only does * have a higher precedence than +, * is to the left of + as well. So in any case, the compiler would evaluate **PI * Radius** before adding 7 to the result. The parentheses make it immediately obvious what operation is to be done first, without having to think back to precedence tables and consider left to right evaluation.

I am something of a fanatic about program readability. Which of the following identical expressions is easier to dope out?

```
R + 2 * PI - 6
(R + (2 * PI)) - 6
```

You have to think a little about the first. You don't have to think about the second at all. I powerfully recommend using parentheses in all but the most completely simpleminded expressions to indicate to all persons, including those not especially familiar with Pascal, the order of evaluation of the operations making up the expression. Parentheses cost you *nothing* in code size or code speed: nothing at all.

To add to your program readability, that's dirt cheap.

12 Statements

Put as simply as possible, a Pascal program is a series of statements. Each statement is an instruction to do something: to define data, to alter data, to execute a function or procedure, to change the direction of the program's flow of control.

There are many different kinds of statements in Pascal. We'll be looking at them all in detail in this chapter and in the next.

Perhaps the simplest of all statements is an invocation of a procedure or function. Turbo Pascal includes several screen-control procedures for moving the cursor around, clearing the screen, and so on. Such procedures are used by naming them, and naming one constitutes a statement:

```
ClrScr;
```

The statement tells the computer to do something, in this case, to clear the screen. The computer executes the statement, the screen is cleared, and control passes to the next statement, whatever that may be.

12.1: ASSIGNMENT AND DEFINITION STATEMENTS

We have been using assignment statements, type definition statements, and variable declaration statements all along. By now you should understand them thoroughly. Type definition statements must exist in the type definition part of a program, procedure, or function. They associate a type name with a description of a programmer-defined type. Enumerated types (see Section 7.6) are an excellent example:

```
TYPE
  Spectrum    = (Red,Orange,Yellow,Green,Blue,Indigo,Violet);
  LongColors  = Red..Yellow;
  ColorSet    = SET OF Spectrum;
```

Each of these three definitions is a statement. The semicolons *separate* the statements rather than terminate them. This is a critical distinction that frequently escapes beginning Pascal programmers. I will take the matter of semicolons up again in Section 13.8.

Variable declaration statements are found only in the variable declaration part of a program, function, or procedure. They associate a variable name with the data type the variable is to have. The colon is used rather than the equal sign:

```
VAR
  I,J,K : Integer;
  Ch    : Char;
  R     : Real;
```

Assignment statements are used in the body of the program, function, or procedure. They copy data from the identifier on the right side of the assignment operator to the variable on the left. With one exception, only a *variable* can be to the left of the assignment operator. Constants, literals, and expressions must be on the right side. Other variables, of course, can also be to the right of the assignment operator:

```
I := 17;                    { '17' is a numeric Literal }
R := PI;                    { PI was defined earlier as a constant }
J := (17*K) + Ord(Ch);      { An expression }
K := J;                     { The value of one variable assigned }
                            { to another }
```

The exception to this rule (under Turbo Pascal 4.0 only) is the **Length** string function, which can be on the left side of an assignment statement as a way of forcing the logical length of a string to some number:

```
Length(MyString) := 42;
```

This is definitely a special case. **Length** and the other string functions will be discussed in Chapter 15.

12.2: TYPE CONFLICTS

Pascal is a strongly typed language. This means that there are rather strict limitations on how data can be moved from a variable of one data type to a variable of a different data type. Strong typing is one way Pascal helps keep nonsense out of your programs.

In Pascal, data items are not merely binary patterns in memory. The type of a data item connotes how that data item should be used. A Boolean variable is more than just a single byte in RAM containing a binary 1 or a binary 0. It carries a value of **True** or **False** which have a logical rather than an arithmetic sense. To attempt to assign a Boolean value to an integer variable simply makes no sense:

```
I := 42;        { Good sense...   }
I := True;      { Nonsense!       }
```

To attempt this kind of assignment statement will trigger:

```
Error #26: Type mismatch.
```

Other languages, notably C, will allow this sort of thing. The code will take whatever binary pattern it finds in one variable and drop it into another variable, whether or not the transfer makes any kind of sense in the context of the program. Without question, keeping sense in a C program is the responsibility of the programmer.

It is easy enough to see why Pascal would try to prevent gross errors in sense like assigning a Boolean value to an integer, and vice versa. Bugs in perfectly sensible programs will prove exasperating enough to locate; trying to debug a program that permits nonsense constructions is an order of magnitude harder.

Compatible and Identical Types

In general, values assigned to a data item must be values with the same data type as the data item. A number of exceptions exist, to cover those situations where assigning one type to another *does* make sense. In such cases, the two types that may be assigned to one another are called "compatible types." For example, types **Integer** and **Byte** are compatible as long as the data they contain fall within the legal range for **Byte**: 0 through 255.

Types are considered identical if they resolve to the same type definition statement. For example, these two variables are *not* type-identical:

```
VAR
  GradeArray : ARRAY[1..6] OF Char;
  LevelArray : ARRAY[1..6] OF Char;
```

Even though they are defined in exactly the same way, their types are assigned in two different statements and therefore are considered by the compiler to be two different types. However, if you were to write:

```
VAR
  GradeArray, LevelArray : ARRAY[1..6] OF Char;
```

then the types of **GradeArray** and **LevelArray** are identical, because they both resolve to the same type definition statement. An even better way to handle the situation is this:

```
TYPE
  GradeStep = ARRAY[1..6] OF Char;

VAR
  Grades : GradeStep;
  Levels : GradeStep;
```

Now, any time you need another variable with an identical type to **Grades** and **Levels**, you can define it using the type **GradeStep**. Whereas all variables of type **GradeStep** refer back to a single type definition statement, the types are considered identical.

Turbo Pascal's type checking is quite strict. Unlike some other Pascals, Turbo's strict type checking cannot be relaxed except for string types that differ in length. Strict type checking requires the use of identical types in assignment statements and in substituting variables into procedure/function parameters (see Section 14.2).

The rules for using compatible types to cross type boundaries are these:

1. Integer types may be assigned to floating point types. There is no problem with this, because any integer may be expressed as a floating point number. The integer 17 becomes the floating point number 17.0. The opposite is not true, however: Floating point numbers may *not* be assigned to integers.
2. Integer types are assignment-compatible with one another. **Byte**, **ShortInt**, **Integer**, **Word**, and **LongInt** may all be freely assigned to one another without type conflict errors. What you *do* have to watch out for is assigning a value from one integer type to a variable of another integer type for which that value is out of range. If you have range checking on, this out-of-range assignment will trigger a range error—**#201**—at runtime. If range checking is turned off, the results will be unpredictable, although, in most cases, the value assigned will simply be truncated to whatever binary pattern will fit into the variable of lesser range. In other words, if integer variable **I** contains a value of 4715, and you assign **I** to variable **B** of type **Byte**, either you will get a range error or the truncated value 107 will be loaded into variable **B**.
3. Subranges of the same base type are compatible. This, for example, is a legal situation:

```
VAR
  Sub1 : 1..10;
  Sub2 : 0..9;

Sub1 := Sub2;
```

This assignment will not trigger a type mismatch error. However, if the value of **Sub2** is 0 when you assign **Sub2** to **Sub1**, a range error will be triggered at runtime if range checking is enabled. It is up to you to make sure that subrange values are range-compatible when assigning two compatible, but not identical, subrange types to each other.

12.3: TRANSFERRING VALUES AMONG INCOMPATIBLE TYPES

Far too many people look at Pascal's strong typing as a brick wall rather than a safety railing. Just as you can climb over a safety railing if you need to, there are ways around *all* of Turbo Pascal's strong typing restrictions if you need to get around them. In this respect, Pascal is every bit as powerful as C, which has little if any type restrictions. It is also considerably safer, especially for people who are just coming up to speed.

Turbo Pascal provides two ways to move values between incompatible types: transfer functions and type casts.

Type Conversion With Transfer Functions

To convert data cleanly between common data types when the conversion makes sense, you can use the standard Pascal transfer functions **Ord**, **Chr**, and **Odd**, **Round**, or **Trunc**. **Ord** changes character data to integer; **Chr** changes integer data to character; and **Odd** changes integer data to Boolean. **Round** and **Trunc** transfer data from real number types to integer types. The individual transfer functions will be discussed in detail in Chapter 16.

Transfer functions "transfer" a value from one type to another, again, only under circumstances in which that transfer makes sense because of some well-defined relationship between the two types involved. For example, all integers are considered either even or odd, depending on whether or not they are divisible by 2. The transfer function **Odd** in effect transfers an integer value to a Boolean value with that even/odd relationship in mind:

```
IsOdd := Odd(I);
```

Here, if **I** contains an odd number, Boolean variable **IsOdd** will be set to **True**. If I contains an even number, **IsOdd** will be given a value of **False**. For every possible integer value, there is a corresponding Boolean value, so the conversion always make sense.

Another example: Given that every ASCII character has a sequential value in the ASCII sort order, the **Ord** function transfers a character value to an integer value:

```
I := Ord(Ch);
```

If **Ch** contained the character **'A'**, then **I** would take on a value of 65, because character **'A'** occupies position 65 in the ASCII sort order.

Range considerations enter into some transfer function conversions. Moving from integer to character is a good example. The transfer function **Chr** returns the character value corresponding to a given integer value, thus transferring a numeric value to a character value:

```
Ch := Chr(I);
```

The problem is that there are only 255 possible character values, but there are 32,767 positive integer values. If I takes on a value of 1000, for example, executing the statement given above will trigger a Turbo Pascal range error:

```
Error 201: Range check error
```

There is no character corresponding to an integer value of 1000, so the conversion in that case doesn't make sense. You, the programmer, are responsible for preventing range errors by testing before making such a conversion:

```
IF (I >= 0) AND (I <= 255) THEN Ch := Chr(I);
```

The same is true with **Round** and **Trunc**. These convert between real numbers and integers, but because real numbers have a larger range than integers, there will always be real numbers whose whole number parts are larger than the largest integer or even long integer. If you try to round a very large real number value into an integer value, you will again trigger a range check error. To prevent range errors, test before you transfer!

Type Conversions With Type Casts

The other, more adventurous method of transferring values between incompatible types is through a process called "type casting." We discussed type casting very briefly in connection with the free union variant record in Section 9.4. A type cast, quite simply, is "pouring" the binary bits that express a value of one type into the same memory space occupied by a variable of another type. This lets the sense of the transfer fend for itself.

As you might expect, the single most important prerequisite for perform type casts is understanding what you're doing on a bits and bytes level. This means studying the way Turbo Pascal's variables are expressed in memory, and how that expression relates to the *meaning* of a value in that variable.

Free union variant records are Standard Pascal's only means of type casting, but this means is by far the most powerful. It can cast any type onto any other type, whether or not the two types are the same size. The implied hazard of casting between two types of differing sizes is avoided, because a free union variant record is always as large as its largest variant part.

For example, consider the following free union variant record:

```
TYPE
  Sneak = RECORD
            CASE Boolean OF
              True:  (KeyChar : Char);
              False: (KeyValue : LongInt)
          END;
```

In essence, what we are doing here is casting a character type on a long integer type and vice versa. Characters are expressed in only 1 byte, while long integers occupy 4 bytes. All that the two types have in common are their first 8 bits, and this free union allows those first 8 bits to be considered as either character type or long integer type.

A variable defined as type **Sneak** is always 4 bytes in size, because its largest variant is a long integer type, which is 4 bytes in size. This is why casting a long integer onto a character through a free union variant record will not overwrite data on either side of the record. You can think of free union **Sneak** as a magic box: Feed it 8 bits of character data, and it will hand you those 8 bits enclosed in a 32 bit long integer "wrapper" with the *high* 24 bits undefined. Or you can hand it 32 bits of long integer data, and it will hand you the low 8 bits of those 32 bits in a character wrapper. Nothing is endangered by the process, except perhaps the sense in extracting only the low 8 bits of a 32-bit numeric type. If within your particular application that extraction makes sense, then so be it: Pascal gives you the means.

Free union variant records are Standard Pascal's only type casting facility, but Turbo Pascal gives you two other mechanisms for performing type casts: value type casting and variable type casting. Whether or not you use them, again, depends on how portable your Pascal application must be. If portability is not a consideration, value and variable type casts are terser and easier to understand than free union variant records.

A *value* type cast changes the type of an expression. The syntax is very much like that of a transfer function:

```
TypeToConvertTo(<expression>)
```

The type may be any legal Pascal type, either predefined or programmer-defined. The value type cast is itself an expression, and may be used anywhere an expression may be used. By the same token, as it is, in fact, an expression, it may *not* be used on the left side of an assignment statement. For that you need a variable type cast, as I'll describe later. For example, to convert the address of a variable to a long integer, you could use this value type cast:

```
MyLong := LongInt(@X);
```

The expression **@X** returns a generic pointer to variable **X**. This cast pours the 32 bits contained in a pointer into the 32 bits of a long integer. Whether or not that makes sense is up to you, but the transfer is done safely and consistently.

Value type casts may be between types of differing sizes. If you cast a long type onto a shorter type, the long type will be truncated to the size of the short type, and data may well be lost. If you have range checking enabled, a range error will usually be generated if data is truncated in a cast between a long type and a short type.

For example, this syntactically legal type cast between a **Char** and a **LongInt** will generate a range error if the **LongInt** has a value greater than 255:

```
Ch := Char(MyLong);
```

In almost all cases, a type cast simply maps the bits of one type into the bits of another type. There are some exceptions, one of them being *sign extension* when casting between signed types. In a value type cast, the sign of the expression being cast is preserved, or *extended* as it is called, to the resulting value if the type converted to has a sign. For example, in casting between a **ShortInt** and a **LongInt**, which are both signed types, the sign bit of the **ShortInt** is not mapped into a data bit in the **LongInt**, but instead becomes the sign bit of the **LongInt**. That way, a negative **ShortInt** will be mapped into a negative **LongInt**:

```
MyLong := LongInt(MyShort);
```

Here, if **MyShort** takes on a value of −17, it will cast onto a **LongInt** value of −17, even though bit 7 of the **MyShort** would ordinarily map onto bit 7 (a data bit) of **MyLong**. Sign extension moves **MyShort**'s sign bit out to bit 31 of **MyLong**, where long integers keep their sign.

Note that a range error is not possible when casting a **ShortInt** onto a **LongInt** but is definitely possible when casting a **LongInt** onto a **ShortInt**. Furthermore, sign extension does not operate through an unprotected range error. In other words, if you turn off range checking and truncate a **LongInt** value into a **ShortInt**, the sign of the **LongInt** will *not* be extended into the **ShortInt**:

```
MyShort := ShortInt(MyLong);
```

In this example, turning off range checking and casting an out-of-range value of −1704 onto the **ShortInt** will *not* preserve the sign. The **ShortInt** will instead take on the value 88.

Variable Type Casts

Somewhat more subtle are Turbo Pascal's variable type casts. Variable type casts operate *only* on variables. What a variable type cast does is temporarily change the type of a variable to the type of the cast. This is done within the restriction that the two types must be the *same physical size* as measured by the **SizeOf** function.

In other words, you can perform a variable type cast between a byte and a short integer, but not between a byte and a long integer or a byte and a pointer. This restriction is what protects from damage variables adjacent in memory to a variable being cast upon. The source and the destination must be the same size.

Another difference from value type casts is that a variable type cast may be used anywhere a variable may be used, including the left side of an assignment statement:

```
Pointer(MyLong) := @FillBuffer;
```

Here, a pointer value returned by the address-of function @ is cast onto a long integer variable. Because pointers and long integers are both 32-bit types, a variable type cast is allowed.

The notation becomes a little more interesting when arrays and records are involved. Consider the following types:

```
TYPE
  FourBytes = ARRAY[0..3] OF Byte;
  TagRecord = RECORD
                Tag1 : Byte;
                Tag2 : ShortInt;
                Tag3 : Char;
                Tag4 : Boolean
              END;
```

Because these two types are both 4 bytes in size, variables of their types may be cast freely onto one another using variable type casts. Individual elements within an array are accessed by placing the index in brackets. Individual fields within a record are accessed by "dotting" to the desired field. These qualifiers are fully supported during variable type casts:

```
VAR
  TestArray : FourBytes;
  TestRecord : TagRecord;


FourBytes(TestRecord)[2] := 65;

TagRecord(TestArray).Tag4 := False;
```

These assignment statements bear some close examination. The first statement treats **TestRecord** as an array for this assignment. Element 2 of a **FourBytes** array corresponds to the **Tag3** field of a **TagRecord**. Assigning the number 65 to element 2 of a **FourBytes** array cast onto a **TestRecord** essentially places the letter A into field **Tag3** of the **TestRecord**. Can you see why? If not, imagine that you have the structures of the two data types mapped out onto transparent plastic sheets, and you overlay one sheet upon another. This is an excellent metaphor for any type cast. Element 2 of a **FourBytes** array overlays **Tag3** of a **TagRecord**.

In a similar fashion, assigning a Boolean value of **False** to the **Tag4** field of a **TagRecord** cast onto a **FourBytes** array places a value of 0 (the internal representation of Boolean **False**) into element 3 of the array. Think of the platic overlays again: the **Tag4** field and element 3 of the array are both the last byte in their respective types. The same bit pattern in that byte will be interpreted as a Boolean **False** value in **Tag4** and as a **Byte** numeric value of 0 in element 3 of the array.

Again, type casting is difficult stuff that you shouldn't attempt until you understand thoroughly how Turbo Pascal represents all its various types in memory and how those types contribute to the meaning of Pascal statements.

13 Controlling Program Flow

Controlling the flow of program logic is one of the most important facets of any programming language. Conditional statements that change the direction of the flow of control, looping statements that repeat some action a number of times, switch statements that pick one course out of many based on a controlling value: All these make useful programs possible.

Furthermore, Pascal would like you, the programmer, to direct the flow of control in a structured, rational manner so that the programs you write are easy to read and easy to change when they need changing. For this reason, wild-eyed zipping around inside a program is difficult in Pascal.

The language syntax itself forcefully suggests that programs begin at the top of the page, progress generally downward, and exit at the bottom when the work is done. Multiple entry and exit points and unconditional branching via **GOTO** are almost literally more trouble than they are worth.

This section examines those statements that channel the flow of program logic: **IF/THEN/ELSE, FOR/DO, WHILE/DO, REPEAT/UNTIL, CASE OF,** and that rascal **GOTO**.

13.1: BEGIN AND END

We have already looked at several simple types of statements, such as assignment statements and data definition statements. In Pascal, you frequently need to group a number of statements together and treat them as though they were a single statement. The means to do this is the pair of reserved words **BEGIN** and **END**. A group of statements between a **BEGIN** and **END** pair becomes a *compound statement*. The bodies of procedures and functions, and of programs themselves, are compound statements:

```
PROGRAM Rooter;

VAR
  R,S : Real;

BEGIN
  Writeln('>>Square root calculator<<');
  Writeln;
  Write('>>Enter the number: ');
  Readln(R);
  S := Sqrt(R);
  Writeln('  The square root of ',R:7:7,' is ',S:7:7,'.')
END.
```

The statements bracketed by **BEGIN** and **END** in the above example are all simple statements, but that need not be the case. Compound statements may also be parts of larger compound statements:

```
PROGRAM BetterRooter;

VAR
  R,S : Real;

BEGIN
  Writeln('>>Better square root calculator<<');
  Writeln;
  R:=1;
  WHILE R<>0 DO
    BEGIN
      Writeln('>>Enter the number (0 to exit): ');
      Readln(R);
      IF R<>0 THEN
      BEGIN
        S := Sqrt(R);
        Writeln('  The square root of ',R:7:7,' is ',S:7:7,'.');
        Writeln
      END
    END;
  Writeln('>>Square root calculator signing off...')
END.
```

This program contains two compound statements, one nested inside the other, and both nested within the compound statement that is the body of the program itself.

This is a good place to point out the prettyprinting convention that virtually always is used when writing Pascal code. The rule on prettyprinting turns on compound statements: *Each compound statement is indented two spaces to the right of the rest of the statement in which it is nested.*

It's also crucial to remember that *prettyprinting is ignored by the compiler.* It is strictly a typographical convention to help you sort out nested compound statements by "eyeballing" rather than by counting **BEGIN**s and **END**s. As some programmers do, you could just as well indent by three or more spaces instead of two. You could also not indent at all. The compiler doesn't care, but the readability of your program will suffer if you don't use prettyprinting.

There is one other thing about compound statements that might seem obvious to some but very unobvious to others: *Compound statements can be used anywhere a simple statement can.* Anything between a **BEGIN/END** pair is treated syntactically by the compiler as just a single simple statement would be.

13.2: IF/THEN/ELSE

A conditional statement is one that directs program flow in one of two directions based on a Boolean value. In Pascal, the conditional statement is the **IF/THEN/ELSE** statement. The **ELSE** clause is optional, but every **IF** reserved word *must* have a **THEN** associated with it. In its simplest form, such a statement is constructed this way:

```
IF <Boolean expression> THEN <statement>
```

The way this statement works is almost self-explanatory in the English language: If the Boolean expression evaluates to **True**, then **<statement>** is executed. If the Boolean expression evaluates to **False**, control "falls through" to the next statement in the program.

Adding an **ELSE** clause makes the statement look like this:

```
IF <Boolean expression> THEN <statement1>
  ELSE <statement2>
```

Here, if the expression evaluates to **True**, then **<statement 1>** is executed. If the expression evaluates to **False**, then **<statement 2>** is executed. If an **ELSE** clause exists, you can be sure that one or the other of the two statements will be executed.

Either one or both of the statements associated with **IF/THEN/ELSE** may be compound statements. Remember that a compound statement may be used anywhere a simple statement may. For example:

```
IF I < 0 THEN
  BEGIN
    Negative := True;        { Set negative flag     }
    I := Abs(I)              { Take abs. value of I  }
  END                        { Never semicolon here! }
ELSE Negative := False;      { Clear negative flag   }
```

An important point to remember: There is *no* semicolon after the **BEGIN/END** compound statement. The entire code fragment above is considered a single IF statement. A crucial corollary: Within an **IF** statement, there is *never* a semicolon immediately before an **ELSE** reserved word. Adding a semicolon will give you a "freestanding ELSE," which is meaningless in Pascal and will trigger a syntax error.

Nested IF Statements

Because an IF statement is itself a perfectly valid statement, it may be one or both of the statements contained in an **IF/THEN/ELSE** statement. **IF**s may be nested as deeply as you like, but remember that if someone reading your code must dive too deeply to find the bottommost **IF**, he or she may lose track of things and drown before coming up again. If the sole purpose of multiply-nested **IF**s is to choose one alternative out of many, it is far better to use the **CASE OF** statement, which will be covered in Section 13.3. Structurally, such a construction looks like this:

```
IF <Boolean expression1> THEN
  IF <Boolean expression2> THEN
    IF <Boolean expression3> THEN
      IF <Boolean expression4> THEN
        IF <Boolean expression5> THEN
          <statement>;
```

The bottom line here is that all Boolean expressions must evaluate to **True** before **<statement>** is executed.

Such a "down escalator" of **IF**s is often hard to follow. Sharp readers may already be objecting that this same result could be done with **AND** operators:

```
IF <Boolean1> AND <Boolean2> AND <Boolean3>
  AND <Boolean4> AND <Boolean5> THEN
  <statement>;
```

This is equivalent to the earlier nested **IF**, but with one sneaky catch: Here, *all* the Boolean expressions are evaluated before a decision is reached on whether or not to execute **<statement>**. With the nested **IF**, the compiler will stop testing as soon as it encounters a Boolean expression that turns up **False**. In other words, in a nested **IF**, if **<Boolean3>** is found to be **False**, the compiler never even evaluates **<Boolean4>** or **<Boolean5>**.

Nitpicking? No! There are times when, in fact, the reason for **<Boolean3>** might be to make sure **<Boolean4>** is not tested in certain cases. Divide by 0 is one of those cases. Consider this:

```
IF AllOK THEN
  IF R > PI THEN
    IF S <> 1 THEN
      IF (R / ((S*S)-1) < PI) THEN
        CalculateRightAscension;
```

Here, a value of **S = 1** will cause a divide-by-zero error if the code attempts to evaluate the next expression. The code *must* stop testing if **S > 1** turns up **False** or risk crashing the program with a runtime divide-by-zero error.

With nested **IF**s, you can determine the sequential order in which a series of tests is done. A string of **AND** operators between Boolean expressions may evaluate those expressions in any order dictated by the code generator's optimization logic. If one of your tests in a complicated expression carries the hazard of a runtime error, use nested **IF**s.

Nested ELSE/IFs

The previous discussion of nested **IF**s did not include any **ELSE** clauses. Nesting **IF**s does not preclude **ELSE**s, although the use and meaning of the statement change radically. Our previous example executed a series of tests to determine whether or not a single statement was to be executed. By using nested **ELSE/IF**s, you can determine which of many statements is to be executed:

```
IF <Boolean1> THEN <statement1>
  ELSE
    IF <Boolean2> THEN <statement2>
      ELSE
        IF <Boolean3> THEN <statement3>
          ELSE
            IF <Boolean4> then <statement4>
              ELSE <statement5>;
```

The code will descend the escalator. As soon as it finds a Boolean with a value of **True**, it will execute the statement associated with that Boolean. The tests are performed in order, and, even if **<Boolean4>** is **True**, it will not be executed (or **<Boolean4>** even evaluated) if **<Boolean2>** is found to be **True** first.

The final **ELSE** clause is not necessary; it provides a *none of the above* choice in case none of the preceding Boolean expressions turns out to be true. You could simply omit it and control would fall through to the next statement without executing any of the statements contained in the larger IF statement.

As with nested **IF**s described above, nested **ELSE/IF**s allow you to set the order of the tests performed. Thus, if one of them carries the danger of a runtime error, you can defuse the danger with an earlier test for the dangerous values.

The **CASE OF** statement is a shorthand form of nested **ELSE/IF**s in which all of the Boolean expressions are of this form: **<value>** = **<value>** and the type of all **<value>**s is identical. We'll look at the **CASE OF** statement in Section 13.3.

Short-Circuit Boolean Evaluation

Apart from nesting **IF** statements, there is another, considerably less portable means of dictating the order in which the compiler evaluates Boolean expressions. It involves a compiler optimization technique new to Turbo Pascal 4.0 called "short-circuit Boolean evaluation."

On page 150 we looked at a nested **IF** construction that avoided the possibility of a divide-by-zero error by testing for a divide-by-zero condition before performing the actual divide operation. Consider that nested **IF** expressed as a single Boolean expression:

```
IF AllOK    AND
   (R > PI) AND
   (S <> 1) AND
   (R / ((S*S)-1) < PI)       { Could trigger divide-by-zero! }
THEN CalculateRightAscension;
```

In most cases, the compiler will evaluate all four of the Boolean subexpressions before combining them into a single Boolean value as the result of the entire, larger expression. As we mentioned earlier, if **S** ever takes on a value of 1, the fourth subexpression will trigger a divide-by-zero error, because when **S** equals 1, **R** is divided by **((S*S)-1)**, which equals zero.

By turning on short-circuit Boolean evaluation, the compiler is forced to evaluate Boolean expressions from left to right. It will stop evaluation as soon as it is sure that testing further will not change the ultimate value of the expression.

How can it be sure that further testing won't change things? Think about the meaning of the **AND** operator. If any number of Boolean subexpressions are **AND**ed together, *a single* ***FALSE*** *value will force the whole expression to* ***FALSE***. Therefore, once that first **FALSE** turns up in the evaluation of an expression from left to right, the whole expression will be **FALSE**, no matter what else waits to be evaluated further to the right.

This technique is called "short circuit" because it quits, when possible, before evaluating an entire expression. Its primary function is to make your programs run more quickly if there are many Boolean expressions to be evaluated. Any time you can perform the same job by executing less code, the job will go more quickly.

However, there is a more generally useful side benefit—allowing you to arrange your Boolean expressions in such a way that "dangerous" subexpressions are to the right of *sentinel* subexpressions, which guard against the dangerous condition. Returning to the example above: If short-circuit Boolean evaluation is enabled, the compiler will evaluate the subexpression **(S <> 1)**. If **S** is equal to 1, evaluation stops there, and the dangerous expression **(R / ((S*S)−1) < PI)** will never be evaluated, which eliminates the possibility of a divide-by-zero runtime error that can bring the program to a halt.

Short-circuit evaluation doesn't only apply to the **AND** operator. A string of expressions **OR**ed together will be evaluated *only* until the first **TRUE** value is encountered. More complicated expressions are simply ground through until the compiler is certain that nothing further can change the ultimate Boolean value. Then it will stop.

Using short-circuit Boolean evaluation is made easier by the fact that the compiler assumes it by default. What is called *complete* Boolean expression evaluation must be explicitly chosen if you want to use it. The Turbo Pascal 3.0 compiler and earlier compilers *always* used complete expression evaluation, which is in keeping with the Standard Pascal definition. Forcing complete Boolean expression evaluation is done from the Options menu in the programming environment, or through the **/B+** switch when using the command-line version of the compiler.

You can also force complete Boolean expression evaluation by inserting a **{$B+}** compiler command in your source code. By bracketing a region of code between a **{$B+}** command and a **{$B−}** command, you can force complete evaluation *only* between the two commands and use short-circuit evaluation throughout the rest of your program.

Why would you ever want to use complete Boolean expression evaluation? Just as you sometimes need to ensure that a certain subexpression will *never* be evaluated, you must sometimes ensure that *every* subexpression is *always* evaluated.

Almost all such cases involve Pascal functions that return Boolean values after doing some sort of necessary work that must be completed regardless of which value the function returns. (For those of you reading this book serially who may not yet understand Pascal functions and how they return values, look ahead to Chapter 14.)

```
IF AllocateBigBuffer(BigBuffPtr) AND
  AllocateLittleBuffer(LittleBuffPtr)
    THEN LoadBothBuffers ELSE
      BEGIN
        IF BuffPtr1 <> NIL THEN LoadBuffer1
          ELSE IF BuffPtr2 <> NIL THEN LoadBuffer2
            ELSE UseDiskSwap := True
      END;
```

This example comes from a program that needs a lot of memory for buffers. It attempts to allocate both its large and its small buffers if possible. If only the large buffer can be allocated without leaving enough RAM for the small buffer, so be it. Or, if the large buffer cannot be allocated but the small buffer can, the small buffer will be allocated and used. Finally, if the program can't find enough memory to allocate either RAM-based buffer, it will use a disk-swapping system to make disk space serve as memory space, albeit slower, for buffer operations. **BuffPtr1** and **BuffPtr2** are pointers that are initialized to point to their buffers when those buffers are allocated. If there isn't enough memory to allocate a buffer, its pointer is returned with a value of **NIL** to indicate that its buffer could not be created.

Whether or not both buffers can be created in memory, *both* pointers must be initialized to some value, either a legitimate pointer value to an allocated buffer, or else **NIL**. Later in the program, the logic will test those pointers to see if a given buffer exists. Having either pointer in an uninitialized state could be disastrous.

If we allowed short-circuit Boolean expression evaluation here, function **AllocateLittleBuffer** would never be executed if **AllocateBigBuffer** failed to find enough memory to allocate the large buffer. **LittleBuffPtr** would be left in an uninitialized state, and later on, the program could malfunction if it tried to test the uninitialized **LittleBuffPtr**.

In this situation, the entire Boolean expression

```
IF AllocateBigBuffer(BigBuffPtr) AND
  AllocateLittleBuffer(LittleBuffPtr)
```

must be evaluated, regardless of the outcome of executing function **AllocateBigBuffer**. The only way to guarantee this is to force complete Boolean expression evaluation, either by issuing a command to the environment or command-line compiler, or by bracketing this area of code between **{$B+}** and **{$B−}** commands.

As the somewhat specialized and arcane nature of this example suggests, short-circuit Boolean evaluation will be your method of choice in most instances.

13.3: CASE OF

Choosing between one of several alternative control paths is critical to computer programming. We've seen how **IF/THEN/ELSE** in its simplest form can choose between two alternatives based on the value of a Boolean expression. By nesting **IF/THEN/ELSE** statements one within another, we can choose among many different control paths, as we saw in the previous section.

The problem of readability appears when we nest more than two or three **IF** statements. Nested **IF/THEN/ELSE** gets awkward and nonintuitive in a great hurry when more than three levels exist. Consider the problem of flashing a message on a CRT screen based on a particular input code number. A problem reporting system on a CRT-equipped computerized car might include a statement sequence like this:

```
Beep;
Writeln('*****WARNING*****');
IF ProblemCode = 1 THEN
  Writeln('[001] Fuel supply has fallen below 10%')
ELSE IF ProblemCode = 2 THEN
  Writeln('[002] Oil pressure has fallen below min spec')
ELSE IF ProblemCode = 3 THEN
  Writeln('[003] Engine temperature is too high')
ELSE IF ProblemCode = 4 THEN
  Writeln('[004] Battery voltage has fallen below min spec')
ELSE IF ProblemCode = 5 THEN
  Writeln('[005] Brake fluid level has fallen below min spec')
ELSE IF ProblemCode = 6 THEN
  Writeln('[006] Transmission fluid level has fallen below min spec')
ELSE IF ProblemCode = 7 THEN
  Writeln('[007] Radiator water level has fallen below min spec')
ELSE
  Writeln('[***] Logic failure in problem reporting system')
```

This will work well enough, but it takes some picking through to follow it clear to the bottom. This sort of selection of one statement from many based on a single selection value is what the **CASE OF** statement was created for. Rewriting the above statement with **CASE OF** gives us this:

```
Beep;
Writeln('*****WARNING*****');
CASE ProblemCode OF
  1 : Writeln('[001] Fuel supply has fallen below 10%');
  2 : Writeln('[002] Oil pressure has fallen below min spec');
  3 : Writeln('[003] Engine temperature is too high');
  4 : Writeln('[004] Battery voltage has fallen below min spec');
  5 : Writeln('[005] Brake fluid level has fallen below min spec');
  6 : Writeln('[006] Transmission fluid level is below min spec');
  7 : Writeln('[007] Radiator water level has fallen below min spec')
ELSE
  Writeln('[***] Logic failure in problem reporting system')
END; { CASE }
```

Here, **ProblemCode** is called the "case selector." The case selector may be an expression or simply a variable. It holds the value upon which the choice among statements will be made. The numbers in a line beneath the word **CASE** are called "case labels." Each case label is followed by a colon and then a statement. The statement may be simple, as in the example, or compound.

When the **CASE** statement is executed, the case selector is evaluated and its value is compared, one by one, against each of the case labels. If a case label is found that is equal to the value of the case selector, the statement associated with that case label is executed. Once the statement chosen for execution has completed executing, the work of the **CASE OF** statement is done and control passes on to the rest of the program. Only one (or none, see below) of the several statements is executed for each pass through the **CASE OF** statement.

If no case label matches the value of the case selector, the statement following the **ELSE** keyword is executed. **ELSE** is optional, by the way. If no **ELSE** keyword is found, control falls through to the next statement in the program.

The general form of a **CASE OF** statement is this:

```
CASE <case selector> OF
  <constant list 1> : <statement 1>;
  <constant list 2> : <statement 2>;
  <constant list 3> : <statement 3>;
                ....
  <constant list n> : <statement n>
  ELSE <statement>
END;
```

There may be as many case labels as you like, up to 256. You may be puzzling over the fact that what we pointed out as case labels are called "constant lists" in the general form. In our first example, each case label was only a single numeric constant. A case label may also be a list of constants separated by commas. Remember that the case label is the *list* of constants associated with a statement; each statement can only have *one* case label. Also, do not forget that a case label may *never* be a variable!

For another example, let's look into part of the code for a mail-in questionnaire analysis system. The responses are to be grouped together by geographical regions of the country. **State** is an enumerated type including all the two-letter state name abbreviations, in alphabetical order. This particular code fragment tallies the number of responses from each geographical region:

```
TYPE
  {II = Indiana; OG = Oregon to avoid reserved word conflict}
  State = (AK,AL,AR,AZ,CA,CO,CT,DE,DC,FL,GA,HI,ID,IL,II,
           IA,KS,KY,LA,MA,MD,ME,MI,MN,MO,MS,MT,NE,NV,NH,
           NJ,NM,NY,NC,ND,OH,OK,OG,PA,RI,SC,SD,TN,TX,UT,
           VA,VT,WA,WI,WV,WY)

VAR
  FromState : State;

CASE FromState OF
CT,MA,ME,NH,
RI,VT           : CountNewEngland := CountNewEngland + 1;
DC,DE,MD,NJ,
NY,PA           : CountMidAtlantic := CountMidAtlantic + 1;
FL,GA,NC,SC     : CountSoutheast := CountSoutheast + 1;
IA,IL,II,MI,
MN,OH,WI,WV     : CountMidwest := CountMidwest + 1;
AL,AR,KY,LA,
MO,MS,TN,VA     : CountSouth := CountSouth + 1;
KS,ND,NE,SD,
WY              : CountPlains := CountPlains + 1;
AK,CA,CO,HI,
ID,MT,OG,UT,
WA,             : CountWest := CountWest + 1;
AZ,NM,NV,OK,
TX              : CountSouthwest := CountSouthwest + 1;
END; { CASE }
```

Here you can see that a case label can indeed be a list of constants. Also note that there is no **ELSE** clause here, because every one of the possible values of type **State** is present in one of the case labels.

CASE OF Cautions

The most important thing to remember about case labels is that they must be constants or lists of constants. A particular value may appear only once in a **CASE OF** statement. In other words, the value **IL** from the last example could not appear in both the **CountMidwest** and the **CountSouth** case labels. The reason for this should be obvious: If a value is associated with more than one statement, the **CASE OF** logic will not know which statement to execute for that case label value.

You should be careful when using a case selector of type **Integer.** Case selectors may only have an ordinality between 0 and 255. An integer case selector may have a value much larger than 255, and, when it does, the results of executing the **CASE OF** statement are undefined. If you work a lot with numeric codes and intend to use **CASE OF** structures to interpret those codes, it is a *very* good idea to define those codes as subranges of **Integer**:

```
TYPE
  Keypress = 0..255;
  Problem  = 0..32;
  Priority = 0..7;
```

Any of these named subrange types may act as case selectors.

ISO Standard Pascal Lacks ELSE in CASE OF

I should point out an important variance between Turbo Pascal and ISO Standard Pascal.

One of the puzzling lapses of logic in ISO Standard Pascal is the lack of an **ELSE** clause in its definition of **CASE OF**. In Standard Pascal, a case selector value for which no case label exists is supposed to cause a runtime error. The programmer is supposed to ensure, with range testing, that each value submitted to a **CASE OF** statement is in fact legal for that **CASE OF** statement. No good explanation as to why this should be necessary has ever crossed my desk.

Therefore, it is not surprising that every single commercial implementation of Pascal that I have tested includes an **ELSE** clause in its definition just to cover that "none of the above" possibility. Turbo Pascal uses the reserved word **ELSE** for this purpose. A number of Pascal compilers (such as UCSD Pascal and Turbo Pascal for the Macintosh) use the keyword **OTHERWISE** instead of **ELSE**, but the meaning and function are exactly the same.

13.4: FOR LOOPS

There are many occasions when you must perform the same operation or operations on a whole range of values. The example used most would be the generation of a square root table for all the numbers from 1 to 100. Pascal provides a tidy way to loop through the same code for each value, dropping through to the next statement in the program when all the loops have been performed. It's called the **FOR** statement, and it is one of three ways to perform program loops in Pascal.

Printing the table of square roots becomes easy:

```
FOR I := 1 TO 100 DO
  BEGIN
    J := Sqrt(I);
    Writeln('The square root of ',I,' is ',J)
  END;
```

There are better ways to lay out a square roots table, but this gets the feeling of a **FOR** loop across very well. **I** and **J** are integers. The compound statement between the **BEGIN/END** pair is executed 100 times. The first time, **I** has the value 1. Each time the compound statement is executed, the value of **I** is increased by 1. Finally, after the compound statement has been executed with the value of **I** as 100, the **FOR** statement has done its job and control passes on to the next statement in the program.

The preceding example is only one particular case of a **FOR** statement. The general form of a **FOR** statement is this:

```
FOR <control variable> :=
  <start value> TO <end value> DO <statement>
```

Here, **<start value>** and **<end value>** may be expressions, and **<control variable>** is any ordinal type, including enumerated types. When a **FOR** statement is executed, the following things happen: If **<start value>** and **<end value>** are expressions, they are evaluated and tucked away for reference. Then, the control variable is assigned **<start value>**. Next, **<statement>** is executed. After **<statement>** is executed, the *successor value* to the value already in the control variable is placed in the control variable. The control variable is now tested. If it exceeds **<end value>**, execution of the FOR statement ceases. Otherwise **<statement>** is executed.

The loop repeats until the control variable is incremented past **<end value.>**

Note that the general definition of a **FOR** statement does not speak of "adding 1 to" the control variable. The control variable is incremented by assigning the successor value of the current value to the control variable. Adding is not really done at all, not even with integers. To obtain the successor value, the statement evaluates the expression:

```
Succ(<control variable>)
```

The function **Succ** is discussed in Section 16.6. If you recall our enumerated type **Spectrum**:

```
TYPE
  Spectrum = (Red, Orange, Yellow, Green, Blue,
              Indigo, Violet);
```

the successor value to **Orange** is **Yellow**. The successor value to **Green** is **Blue**, and so on.

A variable of type **Spectrum** makes a good control variable in a FOR loop:

```
LightSpeed := 3.0E06;
FOR Color := Red TO Violet DO
  Frequency[Color] := LightSpeed / Wavelength[Color];
```

So do characters:

```
FOR Ch := 'a' to 'z' DO
  IF Ch IN CharSet DO Writeln('CharSet contains ',Ch);
```

If **<start value>** and **<end value>** are the same, the loop is executed once. If **<start value>** is *higher* than **<end value>**, the loop is not executed at all. That is, **<statement>** is not executed, and control immediately falls through to the next statement in the program.

Control Variable Cautions

A control variable must be an ordinal type or a subrange of an ordinal type. Real numbers cannot be used as control variables.
There is no distinct successor value to a number like 3.141592, after all. For similar reasons, you cannot use sets or structured types of any kind.

Also, an error message will appear if the control variable is a formal parameter passed by reference (see Section 14.2); that is, a **VAR** parameter in the function or procedure's parameter line. You cannot do this:

```
PROCEDURE Runnerup(Hi,Lo     : Integer;
                   VAR Limit : Integer);

VAR
  Foo : Integer;

BEGIN
  <statements>;
  FOR Limit := Lo TO Hi DO <statement>;
  Foo:=Limit;
  <statements>
END;
```

Turbo Pascal will respond here with:

```
Error 97: Invalid FOR control variable
```

Standard Pascal requires that control variables be local and nonformal. Turbo Pascal is somewhat more lenient, and allows control variables to be nonlocal (i.e., not declared in the current block). It also allows formal parameters to be control variables as long as they are passed by value, that is, if the procedure is given its own copy of the parameter to play with.

As in many of Pascal's rules and restrictions, this one was designed to keep you out of certain kinds of trouble. Understanding what kind of trouble requires a little further poking at the notion of control variables in **FOR** loops: To make procedure **Runnerup** work, some sort of local control variable would have to be declared in the data declaration section of the procedure:

```
PROCEDURE Runnerup(Hi,Lo      : Integer;
                   VAR Limit : Integer);
VAR
  Foo,I : Integer;

BEGIN
  <statements>;
  FOR I := Lo TO Hi DO <statement>;
  Foo := I;
  <statements>
END;
```

Now the **FOR** loop will compile correctly. There is still something wrong with this procedure, though. What it's trying to do is to make use of the control variable immediately after the loop has executed by assigning its value to **Foo**. This also is illegal. *Immediately after a **FOR** statement, the value of the control variable becomes undefined.* This is not a problem, because the end value is accessible in **Hi**. To make use of the final value of the control variable, assign **Hi** to **Foo** instead of **I**. The end result will be the same:

```
PROCEDURE Runnerup(Hi,Lo      : Integer;
                   VAR Limit : Integer);

VAR
  Foo,I : Integer;

BEGIN
  <statements>;
  FOR I := Lo TO Hi DO <statement>;
  Foo := Hi;
  <statements>
END;
```

There is a good reason for the control variable becoming undefined after its loop has run its course. After each pass through **<statement>**, the successor value to the current value in the control variable is computed. That successor value *may in fact become undefined.* Going back to our type **Spectrum**, what is the successor value to **Violet**? There is none; **Succ(Violet)** is undefined. Consider:

```
FOR Color := Red TO Violet DO <statement>;
```

After the loop runs through its seven iterations, Color would hold an undefined value. If you were allowed to "pick up" the control variable's value after the run of a FOR loop, you might in fact be picking up an undefined, nonsense value and have no way of knowing it were so. So the Standard Pascal definition declares that control variables are *always* undefined after a **FOR** loop to remove the temptation to "save" the final value of a control variable for later use.

In some other Pascal compilers there is a **BREAK** statement that interrupts execution of a **FOR** loop before the loop has run through all the iterations called for. In such compilers you cannot always be sure that the **<end value>** will match the control variable's value at the end of the loop. In those compilers you must use a separate variable to keep track of the value in the control variable. Because Turbo Pascal does not have a **BREAK** statement, this need not concern you.

Within the **FOR** loop (that is, within **<statement>**) the control variable must be treated as a "read-only" value. You *cannot* change the value of the control variable within the FOR loop! The **REPEAT/UNTIL** and **DO/WHILE** statements allow this kind of "moving target" loop, which will execute as often as required to make a control variable equal to some final value. **FOR** loops execute a fixed number of times, and, while executing, the value of the control variable is solely under the control of the **FOR** loop itself.

Now (finally!) you may understand why Turbo forbids using **VAR** parameters as control variables. Pascal reserves the right to force a control variable into an undefined state after its loop is done. Using a **VAR** parameter as a control variable might "reach up" out of the procedure and force the **VAR** parameter into an undefined state. Allowing a procedure to "undefine" a parameter passed to it is asking for trouble, since it may not be obvious to the calling logic that its parameter may come back undefined.

Preserve sanity in your programs. Keep your **FOR** loop control variables local.

FOR with DOWNTO

The **FOR** statement as we've seen it so far always increments its control variable "upward"; that is, it uses the successor value of the control variable for the next pass through **<statement>**. It is sometimes useful to go the other way: to begin with a high value and count down to a lower value. In this case, the *predecessor* value of the value in the control variable becomes the new control variable value for the next pass through **<statement>**. This predecessor value is calculated with the predefined function **Pred(<control variable>)** (see Section 16.6). Otherwise, its operation is identical to that of **FOR** with **TO**:

```
FOR I := 17 DOWNTO 7 DO <statement>;

FOR Color := Indigo DOWNTO Orange DO <statement>;

FOR Ch := 'Z' DOWNTO 'X' DO <statement>;
```

When using **DOWNTO**, keep in mind that if **<start value>** is *lower* than **<end value>**, **<statement>** will not be executed at all. This is the reverse of the case for **FOR/TO** loops.

13.5: WHILE/DO LOOPS

As we have seen, a **FOR** loop executes a specific number of times, no more, no less. The control variable cannot be altered during the loop. There are many cases in which a loop must run until some condition occurs that stops it. The control variable *must* be altered during the loop, or the loop will just run forever. Pascal offers two ways to build such loops: **WHILE/DO** and **REPEAT/UNTIL**.

The general form of a **WHILE/DO** loop is this:

```
WHILE <Boolean expression> DO <statement.>
```

The **<Boolean expression>** can be an expression that evaluates to a Boolean value, such as I > 17, or it can simply be a Boolean variable. As with **FOR** loops, **<statement>** can be any statement including a compound statement framed between **BEGIN** and **END**.

WHILE/DO loops work like this: The code first evaluates **<Boolean expression>**. If the value of the expression is **True**, then **<statement>** is executed. If the value is **False**, **<statement>** is not executed even once, and control falls through to the next statement in the program.

If **<Boolean expression>** came out **True** the first time, then, after executing **<statement>**, the code goes back and evaluates **<Boolean expression>** again. If it is still true, **<statement>** is executed again. If it evaluates to **False**, the **WHILE/DO** loop ends and control passes on to the next statement in the program.

In short, as long as **<Boolean expression>** is **True**, **<statement>** will be executed repeatedly. Only when the expression comes up **False** will the loop end.

For example:

```
VAR
  pH : Real;

FillTank;             { Fill tank with raw water    }
Take(pH);             { Take initial pH reading     }
WHILE pH < 7.2 DO
  BEGIN
    AddAlkali;        { Drop 1 soda pellet in tank }
    AgitateTank;      { Stir }
    Take(pH);         { Read the pH sensor          }
  END;
```

This snippet of code is a part of a control system for some sort of chemical processing apparatus. All it must do is fill a tank with water, ensuring that the pH of the water is at least 7.2. If the water from the water supply is too acidic (water quality varies widely in some parts of the country) its pH must be brought up to 7.2 before the water is considered useable.

First, the tank is filled. Then, the initial pH reading is taken. The water may in fact be useable from the start, in which case the loop is never executed, but if the water comes up acidic, the loop is executed. A small quantity of an alkali is added, the tank is stirred for a while, and then the pH is taken again. If the pH has risen to 7.2, the loop terminates. If the pH remains too low, the loop is executed again, more alkali is added, and the pH is tested once more. This will continue until the pH test returns a value in the variable pH that is higher than 7.2.

It is crucial to note that an initial pH test was performed. If the variable **pH** were not used before, its value is undefined, and testing it for **True** or **False** will not reflect any real-world meaning. *Make sure the Boolean expression is defined before the WHILE/DO loop is executed!* Every variable in the expression must be initialized somehow before the expression can be trusted.

The most important property of a **WHILE/DO** loop is that its Boolean expression is tested *before* **<statement>** is executed. A corollary to this is that there are cases when **<statement>** will never be executed at all.
Keep this in mind while we discuss **REPEAT/UNTIL** below.

13.6: REPEAT/UNTIL LOOPS

REPEAT/UNTIL loops are very similar to **WHILE/DO** loops. As with **WHILE/DO**, **REPEAT/UNTIL** executes a statement until a Boolean expression becomes **True**. The general form is this:

```
REPEAT <statement> UNTIL <Boolean expression>;
```

It works this way. First, **<statement>** is executed. Then **<Boolean expression>** is evaluated. If it comes up **True**, the loop terminates, and control passes on to the next statement in the program. If **<Boolean expression>** evaluates to **False**, **<statement>** is executed again. This continues until **<Boolean expression>** becomes **True**.

The important fact to notice here is that *<statement> is always executed at least once.* So unlike **WHILE/DO**, you needn't initialize all variables in **<Boolean expression>** before the loop begins. It's quite all right to assign all values as part of executing the loop.

For an example, let's return to the chemical process controller and consider a snippet of code to handle a simple titration. Titration means adding small, carefully measured amounts of one chemical to another while watching for some chemical reaction to go to completion. Usually, when the reaction is complete, the mixture will begin changing color, or will become electrically conductive, or give some other measurable signal.

In Pascal, it might be handled this way:

```
VAR
  Drops    : Integer;
  Complete : Boolean;

Drops := 0;
REPEAT
  AddADrop;                   { Opens valve for 1 drop   }
  Drops := Drops + 1;         { Increment counter        }
  Signal(Complete)            { Read the reaction sensor }
UNTIL Complete;
```

The drops counter is initialized before the loop begins. A drop is added to the test vessel, and the drop counter is incremented by one. Then the reaction sensor is read. If it senses that the reaction has gone to completion, the Boolean variable **Complete** is set to **True**.

Since it takes it least one drop to complete the reaction, this series of events must be done at least once. If the first drop completes the reaction, the loop is performed only once. Most likely, the loop will have to execute many times before the chemical reaction completes and **Complete** becomes **True**. When this happens, **Drops** will contain the number of drops required to complete the reaction. The value of **Drops** might then be displayed on an LED readout or other output device attached to the chemical apparatus.

One interesting thing about **REPEAT/UNTIL** is that the two keywords do double duty if **<statement>** is compound. Instead of bracketing the component statements between **BEGIN** and **END**, **REPEAT** and **UNTIL** perform the bracketing function themselves.

WHILE/DO or REPEAT/UNTIL?

These two types of loops are very similar, and it is sensible to ask why both are necessary. Actually, **WHILE/DO** can accomplish anything **REPEAT/UNTIL** can, with a little extra effort and occasional rearranging of the order of the statements contained in the loop. The titration code could be written this way:

```
Drops := 0;
Complete := False;
WHILE NOT Complete DO
  BEGIN
    AddADrop;
    Drops := Drops +1;
    Signal(Complete)
  END
```

This method requires that **Complete** be set to **False** initially to ensure that it will be defined when the loop first tests it. If **Complete** were left undefined and it happened to contain garbage data that could be interpreted as **True**, (if bit 0 is high, for example) the loop might never be executed, and the code would report that the titration had been accomplished with zero drops of reagent, which is chemically impossible.

Using **REPEAT/UNTIL** would prevent that sort of error in logic. Quite simply: *Use **REPEAT/UNTIL** in those cases in which the loop* must *be executed at least once.* Whenever you write code with loops, always consider what might happen if the loop were never excuted at all, and, if anything unsavory might come of it, make sure the loop is coded as **REPEAT/UNTIL** rather than **WHILE/DO**.

13.7: LABELS AND GOTO

The bane of unstructured languages like BASIC, FORTRAN, and COBOL is freeform **GOTO** branching all over the program body without any sort of plan or structure. Such programs are very nearly impossible to read. The problem with such programs, however, is not the **GOTO**s themselves but bad use of them. **GOTO**s fill a certain need in Pascal, but they have a seductive power and are eminently easy to understand:

```
GOTO 150;
```

and *wham!* you're at line 150. This straightforwardness leads inexperienced programmers (especially those first schooled in BASIC, and today, almost everybody is first schooled in BASIC) to use them to get out of any programming spot that they do not fully understand how to deal with in a structured manner.

I will not caution you, as some people do, never to use a **GOTO** no matter what. I'm about to tell you about a few situations where nothing else will do. What I *will* tell you to do is never use a **GOTO** when something else will get the job done as well or better.

GOTO Limitations in Turbo Pascal

Use of **GOTO**s with Turbo Pascal carries a few limitations. You may **GOTO** a label *within* the current block. This means that you may *not* **GOTO** a label inside another procedure or function, or from within a procedure or function out into the main program code.

You may not **GOTO** a label within a structured statement. In other words, given this **WHILE/DO** statement:

```
WHILE NOT Finished DO
  BEGIN
    Read(MyFile,ALine);
```

```
      IF EOF(MyFile) THEN GOTO 300;
      IF ALine = 'Do not write this...' THEN GOTO 250;
      Write(YourFile,ALine);
      LineCounter := LineCounter + 1;
      250:
    END
  300: Close(MyFile);
```

you could not, from some other part of the block, **GOTO** label **250**. However, assuming that this snippet of code is not part of some larger structured statement, you could, in fact, **GOTO** label **300** from some other part of the current block. Whether or not that would perform any useful function is a good question.

This example is *very* bad practice, but it is included here only to indicate that you cannot branch into the middle of a **WHILE/DO** statement from somewhere outside the statement. Label **250** is accessible only from somewhere between the **BEGIN** and **END** pair.

In general, GOTOs are used to get *out* of somewhere, not to get *in*. Standard Pascal and Turbo Pascal both lack two general looping functions that some other Pascal implementations do have: **BREAK** and **CYCLE**. **BREAK** leaves the middle of a loop and sends control to the first statement after the loop. **CYCLE** stops executing the loop and begins the loop at the top, with the next value of the control variable. Turbo Pascal has neither.

Without **BREAK** and **CYCLE**, **GOTO** provides the only reasonably clean way to get out from inside complicated loops in which a lot is going on and more than one condition value affects exiting from the loop. You can always get out of a loop safely by using the facilities provided by the loop (exiting at the top with **WHILE/DO** and at the bottom with **REPEAT/UNTIL**) but there are cases when to do so involves tortuous combinations of **IF/THEN/ELSE** that might in fact be harder to read than simply jumping out with **GOTO**. But for clarity's sake, always **GOTO** the statement *immediately* following the loop you're exiting. This is what the **Break** statement does.

If a future release of Turbo Pascal includes **BREAK**, you will no longer need to use **GOTO** to get out of loops.

The other, far less frequent use of **GOTO** is to get somewhere else *now*, especially when the code you need to get to is code that handles some impending failure or emergency situation requiring immediate attention. This would tend to come up in system-type software that has direct control over some rather complicated hardware. In 9 years of working with Pascal, I have never had to do this. I suspect that if you ever do, you will know it.

13.8: SEMICOLONS AND WHERE THEY GO

Nothing makes newcomers to Pascal cry out in frustration quite so consistently as the question of semicolons and where they go. There are places where semicolons *must* go, places where it seems not to matter whether they go or not, and places where they cannot go without triggering an error. Worse, it seems at first to have no sensible method to it.

Of course, like everything else in Pascal, placing semicolons *does* have a method to it. Why the confusion? Two reasons:

1. Pascal is a "freeform" language that does not take line structure of the source file into account. Unfortunately, most new Pascal programmers graduate into Pascal from BASIC, which is about as line-oriented a language as ever existed.
2. Semicolons in Pascal are statement *separators*, not statement *terminators*. The difference is crucial and is made worse by the fact that the language PL/1 uses semicolons as statement terminators.

Clarifying these two issues should make semicolon placement Pascal-style second nature.

Freeform Versus Line-Structured Source Code

Pascal source code is *freeform*; that is, the boundaries of individual lines and the positioning of keywords and variables on those lines matter not at all. The prettyprinting customary to Pascal source code baffled me in my learning days until I realized that the compiler ignored it. In fact, the compiler sucks the program up from the disk as though through a drinking straw, in one long line. The following two program listings are utterly identical as far as Turbo Pascal is concerned:

```
PROGRAM Squares;

VAR
  I,J : Integer;

BEGIN
  Writeln('Number     Its square');
  FOR I := 1 TO 10 DO
    BEGIN
      J := I * I;
      Writeln(' ',I:2,'          ',J:3)
    END;
  Writeln;
  Writeln('Processing completed!')
END.
```

```
PROGRAM Squares;VAR I,J : Integer;BEGIN Writeln
('Number     It''s square');FOR I:=1 TO 10 DO
BEGIN J:=I*I;Writeln(' ',I:2,'         ',J:3)
END;Writeln;Writeln('Processing completed!') END.
```

Although the second listing appears to exist in four lines, this is only for the convenience of the printed page; the intent was to express the program as one continuous line without any linefeed breaks at all.

The second listing above is the compiler's eye view of your program source code. You must remember that, although you see your program listing "from a height" as it were, the compiler scans it one character at a time, beginning with the **P** in **PROGRAM** and reading through to the . after **END**. All unnecessary whitespace characters (spaces, tabs, carriage returns, and linefeeds) have been removed as the compiler would remove them. Whitespace characters serve only to delineate the beginnings and endings of reserved words and identifiers, and, as far as the compiler is concerned, one whitespace character of any kind is as good as one of any other kind. Once the compiler "grabs" a word or identifier, literal, or operator, it tosses out any following whitespace characters until it finds a non-whitespace character, indicating that a program element is beginning again.

Semicolons as Statement Separators

Note the compound statement executed as part of the FOR loop:

```
BEGIN J:=I*I;Writeln('  ',I:2,'          ',J:3) END
```

There are two statements here, framed between BEGIN and END. As smart as the compiler may seem to you, it has no way to know where statements start and end unless you tell it somehow. If the ; between **I** and **Writeln** were not there, the compiler would not know for sure if the statement that it sees (so far!) as **J:=I*I** ends there or must somehow continue with **Writeln**.

Note that there is no semicolon after the second statement. There doesn't have to be; the compiler has scanned a **BEGIN** word and knows that an **END** should be coming up eventually. The **END** word tells the compiler unambiguously that the previous statement is over and done with. ***BEGIN and END are not statements.*** They are delimiters, and only serve to tell the compiler that the group of statements between them is a compound statement.

I find it useful to think of a long line of statements as a line of boxcars on a rail siding. Separating each car from the next is a pair of linked couplers. Anywhere two couplers connect is where, if boxcars were program statements, you would need a semicolon. You don't need one at the end of the last car because the last car doesn't need to be separated from anything; behind it is just empty air.

The Null Statement

Why, then, is it legal to have a semicolon after the last statement in a compound statement? This is perfectly all right (and adds to the confusion):

```
BEGIN
  ClrScr;
  J := J + 5;
  IF J > 100 THEN PageEject;
  DoPage;                       { ; not needed here }
END
```

The answer, of course, is that there is a statement after **DoPage**; that statement is the null statement. This might be clearer with the example rewritten this way:

```
BEGIN
  ClrScr;
  J := J + 5;
  IF J > 100 THEN PageEject;
  DoPage;
                    { Null statement here! }
END
```

There is a semicolon between **DoPage** and the null statement but none between the null statement and the **END** word.

The null statement is a theoretical abstraction; it does no work and generates no code, not even as **NOP** (No-Op) instruction. Don't try to use it to pad timing loops! It serves very little purpose other than to make certain conditional statements a little more intuitive and readable. For example:

```
IF TapeIsMounted THEN { NULL } ELSE RequestMount;
```

I find this more readable than the alternative:

```
IF NOT TapeIsMounted THEN RequestMount;
```

but I suspect it is a matter of taste. Note the convention of inserting the comment **{NULL}** wherever you use a null statement. It's like the bandages around the Invisible Man; they keep the guy out of trouble.

Another use of the null statement is in **CASE/OF** statements in which nothing need be done for a selector value:

```
CASE Color OF
  Red    : { NULL };           { No filter needed }
  Orange : InsertFilter(1);    { Density 1        }
  Yellow : InsertFilter(5);    { Density 5        }
  Green  : InsertFilter(11);   { Density 11       }
  ELSE InsertFilter(99)        { Opaque (99)      }
END; { CASE }
```

In some sort of optical apparatus there is a mechanism for rotating a filter in front of an optical path. The density of the filter depends on the color of light being used. No filter is needed for red, and for blue, indigo, or violet the test will not function, and an opaque barrier is moved into the optical path instead of a filter. A null statement is used for the **Red** case label.

Semicolons with IF/THEN/ELSE Statements

More errors are made placing semicolons within **IF/THEN/ELSE** statements than any other kind, I suspect. This sort of thing is fairly common and oh, so easy to do when you're a beginner:

```
IF TankIsEmpty THEN FillTank(Reagent,FlowRate);
  ELSE Titrate(SensorNum,Temp,Drops);
```

The temptation to put a semicolon at the end of a line is strong. In most dialects of BASIC, furthermore, you must put a colon between an **IF** clause and its associated **ELSE** clause.

Semicolons are statement *separators*, though, and the example above is *one single statement.* There is nothing to separate. Remember this rule with regard to placing semicolons in **IF/THEN/ELSE** statements: *Never place a semicolon immediately before an ELSE word in an IF statement!* With that in mind you will avoid 90 percent of all semicolon placement errors.

To make things slightly more confusing, it *is* legal, and sometimes necessary, to place a semicolon before an **ELSE** word in a **CASE OF** statement. If a future release of Turbo Pascal allows the use of the **OTHERWISE** word as an alias for **ELSE** within a **CASE OF** statement, it will become possible to say, simply, *never place a semicolon immediately before ELSE.* In the meantime, keep the null statement in mind and semicolon placement won't seem quite so arbitrary.

13.9: HALT AND EXIT

These two statements are *not* part of Standard Pascal and, in many respects, are not good practice—like **GOTO**, they are easily abused and can make your programs difficult to read and debug. Still, like any good tools, they have some legitimate uses.

Halt

Turbo Pascal provides a means of stopping a program in its tracks from anywhere within the program. **Halt** will terminate any running program and throw you out into

Turbo Pascal's main menu (if you are running within the Turbo Pascal Environment) or into DOS.

Halt is a creature of limited usefulness. If you have the foresight to envision a condition in which a complicated program gets so confused that it cannot continue to function meaningfully, it may be best to call a **Halt** with an appropriate error message. One example overlay involves a situation in which a large application consisting of multiple overlay files discovers that one or more of its overlay files cannot be read or may be missing from disk. This is the sort of anomalous situation from which there is no truly graceful exit. It may be better to print an informative error message and return to the operating system.

In the PC DOS and MS DOS versions of Turbo Pascal V 3.0 and later, The **Halt** statement may take an optional parameter:

```
Halt(ErrorCode : Integer);
```

When **Halt** returns control to DOS, it sets DOS **ERRORLEVEL** to the numeric value in the optional **ErrorCode** parameter.

If you are not familiar with **ERRORLEVEL**, it is a pseudovariable that can be tested by the DOS **IF** batch command in order to perform conditional processing in a DOS batch file. After a program is run from a batch file, some value is set into **ERRORLEVEL**, typically zero. A program can set **ERRORLEVEL** if it wishes, and then the batch file can "branch" at that point, depending on the value set into **ERRORLEVEL** by the program.

Aside from its name, there is nothing hard-coded into **ERRORLEVEL** connected with DOS errors. I use **Halt** with a parameter as a means of building small utilities to extend the usefulness of DOS's batch facility, typically by writing a short program to test some machine condition for which no test exists in ordinary batch commands.

For example, the following program tests whether a monochrome or graphics display is installed in the system and uses the **Monochrome** function, which is described in Section 18.4:

```
PROGRAM IsMono;

{$I MONOTEST.SRC}   {See Section 18.4}

BEGIN
  IF Monochrome THEN Halt(5)
    ELSE Halt(0)
END.
```

If a monochrome screen is present, the program returns a code of 5 to DOS **ERROR-LEVEL**; otherwise it returns a 0.

A batch file can test **ERRORLEVEL** to see which display is installed in the machine:

```
ECHO off
ISMONO
IF ERRORLEVEL 5 GOTO mono
ECHO on
REM    It is a graphics screen.
GOTO exit
:mono
ECHO on
REM    It is a monochrome screen.
ECHO on
:exit
```

Your batch file could use this information to install a different set of CRT drivers for an application program, depending on the type of display adapter installed on the machine.

If you do use **Halt** in anything but the simplest programs like the one above, *highlight its presence with glow-in-the-dark comments!* Generally, Pascal programs begin at the top and end at the bottom. Ducking out in the middle via **Halt** or **Exit** (see below) is nonstandard procedure and can complicate the reading and debugging of your programs.

Exit

Release 3.0 and later releases of Turbo Pascal offer the **Exit** statement. **Exit** jumps out of the current block into the next highest block. In other words, if you execute a Exit within a function or procedure, execution of that function or procedure will cease, and control returns to the statement immediately following the invocation of the function or procedure. If **Exit** is encountered in the main program, Turbo will end the program and return to the Turbo Pascal Environment, if you were working from within the Environment, or into DOS, if the program is a standalone .COM file.

Use **Exit** with care. One of the strengths of the Pascal structure is the assurance that a block of code begins at the top and ends at the bottom. Sprinkling a block with **Exit**s makes code much harder to read and debug.

Interrupting a Program with CTRL-Break

Invariably, you're going to grow impatient with some program's plodding along, and you're going to want to cut the run short. An easy exit inherited from BASIC is simply to press CTRL-Break. If the conditions are correct, your program will terminate, and control will return to DOS, if you were running a standalone .EXE file, or to the Turbo Pascal Environment, if you were running from within the Environment.

What are the correct conditions? A program may be "broken" any time it writes to the CRT *if* a system Boolean variable called **CheckBreak** in the **Crt** unit is set to **True**. To modify **CheckBreak** you must **USE** the **Crt** unit, as **CheckBreak** exists within **Crt**.

If **CheckBreak** is set to **True** (its default state) you'll have a way out of infinite loops, as long as somewhere within the infinite loop the program writes something to the screen. You pay a heavy price in performance, however, because the Turbo Pascal runtime code must check frequently to see if a CTRL-Break has been typed.

It makes sense to leave **CheckBreak** to **True** during development and debugging, when endless loops tend to happen. Once the program has been solidly debugged, you can set **CheckBreak** to **False** and pick up the additional speed.

Finally, here is a note to Turbo Pascal 3.0 users: **CheckBreak** serves the same function as the undocumented system variable **CBreak** in Turbo Pascal 3.0.

14 Procedures and Functions

Some people think looping statements such as **WHILE/DO** and **REPEAT/UNTIL** (and the corresponding lack of need for **GOTO**s) are the touchstone of structured programming. This is not so. At the bottom of it, *structured programming is the artful hiding of details.* The human mind's ability to grasp complexity breaks down quickly unless some structure or pattern can be found in the complexity. I recall, with some embarrassment, spending 6 weeks writing a 1300-line FORTRAN program in high school, and, by the time I wrote the last of it, I no longer remembered how the first part worked. The entire program was a mass of unstructured, undifferentiated detail.

How does one hide details in computer programs? It is done by identifying sequences of code that do discrete tasks, setting each sequence off somewhere, and replacing it by a single word describing (or at least hinting at) the task it does. Such code sequences are properly called "subprograms."

In Pascal, there are two types of subprograms: procedures and functions. Both are sequences of Pascal statements set off from the main body of program code. Both are invoked, and their statements executed, simply by naming them. The only difference between functions and procedures is this: The identifier naming a function has a type and takes on a value when it is executed. The name of a procedure has no type and takes on no value.

Two simple examples: The procedure **ClrScr**, when executed, clears the CRT screen:

```
ClrScr;
```

ClrScr will be discussed along with the other CRT control procedures in Section 18.1. It is a complete statement in itself. Although **ClrScr** has no parameters, a procedure may have any number of parameters if it needs them.

A function, by contrast, is *not* a complete statement. It is more like an expression, which returns a value that must be used somehow:

```
VAR
  Space,Radius : Real;

Radius := 4.66;
Space := Area(Radius);
```

Note that you could *not* simply have put the invocation of **Area** on a line by itself:

```
Area(Radius);
```

This would generate an error at compile-time:

```
Error 122: Invalid variable reference
```

Area calculates the area for the value **Radius**, which is passed to it as a parameter. After it calculates the area for the value passed to it in **Radius**, the area value is taken on by the identifier **Area**, as though **Area** were a variable.

Functions in Turbo Pascal may return values of any ordinal type (**Integer**, **Char**, **Byte**, **Real**, **Boolean**), any enumerated type, a subrange of an ordinal or enumerated type, or any pointer type. Unlike virtually all other Pascal compilers, functions in Turbo Pascal may also return values of type **String.**

Using the **Area** function hides the details of calculating areas. There aren't many details involved in calculating areas, but for other calculations (matrix inversion comes to mind) a function can hide thirty or forty lines of complicated code, or even more. When you're reading the program and come to a function invocation, you can think, "Ah, here's where we invert the matrix," without being concerned about *how* the matrix is actually inverted. At that level in reading the program, the "how" is not important, and those details are best kept out of sight.

The **ClrScr** function illustrates another facet of the hiding of details. The **ClrScr** function clears the CRT screen. How this is done varies widely from computer to computer. The IBM PC requires a software interrupt to clear the screen, whereas the Victor 9000 and certain other less-than-compatible PC compatibles only require that a control character be sent to the system console driver. If you're writing a program that is to run on many different computers, it is best to hide machine-specific details in functions and procedures and put those functions and procedures in a machine-specific library that can be included into your main program code for each specific machine. This way, a single source file can be compiled to run on many different computers without changes, simply by including a different machine-specific library for each different computer.

The Structure of Functions and Procedures

Procedures and functions are, in effect, miniature programs. They can have label declarations, constant declarations, type declarations, variable declarations, and procedure and function declarations as well as the expected code statements. Consider these two entities:

```
PROGRAM HiThere;             PROCEDURE HiThere;

BEGIN                         BEGIN
  Writeln('Hi there!')          Writeln('Hi there!')
END.                          END;
```

The only *essential* differences between a program and a procedure are the keyword **PROGRAM** and the punctuation after the final **END**.

Functions are a little different. A function has a type and takes on a value that it returns to the program logic that invokes it:

```
FUNCTION Area(R : Real) : Real;

CONST
  PI = 3.14159;

BEGIN
  Area := PI * R * R;
END;
```

The type of function **Area** is **Real**. As you can see from the function's single line of code, an expression computing area for the given radius R is evaluated, and the value is assigned to the function's name. Aside from these two distinctions, functions are identical to procedures and are also miniature programs.

14.1: FORMAL AND ACTUAL PARAMETERS

Passing data to procedures and functions can be done two ways: By using global variables that any procedure or function can read from or write to, and through each procedure or function's parameter list. The first method (sometimes called "common," from an old FORTRAN scheme) is a thoroughly bad idea and should be avoided as much as possible. It is good practice to hand a procedure everything it needs through its parameter list.

The following program contains a procedure to draw boxes on the CRT screen with characters, as opposed to true pixel graphics:

```
{--------------------------------------------------------------}
{                          BoxTest                             }
{                                                              }
{                Character box draw demo program               }
{                                                              }
{                          by Jeff Duntemann                   }
{                          Turbo Pascal V5.0                   }
{                          Last update 7/14/88                 }
{                                                              }
{                                                              }
{                                                              }
{--------------------------------------------------------------}

PROGRAM BoxTest;

USES Crt;

TYPE
  GrafRec = RECORD
              ULCorner,
              URCorner,
              LLCorner,
              LRCorner,
              HBar,
              VBar,
              LineCross,
              TDown,
              TUp,
              TRight,
              TLeft : String[4]
            END;

     String80  = String[80];


VAR
  GrafChars    : GrafRec;
  X,Y          : Integer;
  Width,Height : Integer;


PROCEDURE DefineChars(VAR GrafChars : GrafRec);

BEGIN
  WITH GrafChars DO
    BEGIN
      ULCorner  := Chr(201);
      URCorner  := Chr(187);
      LLCorner  := Chr(200);
      LRCorner  := Chr(188);
      HBar      := Chr(205);
      VBar      := Chr(186);
      LineCross := Chr(206);
      TDown     := Chr(203);
      TUp       := Chr(202);
      TRight    := Chr(185);
      TLeft     := Chr(204)
    END
END;


PROCEDURE MakeBox(X,Y,Width,Height : Integer;
                  GrafChars        : GrafRec);

VAR
  I,J : Integer;

BEGIN
  IF X < 0 THEN X := (80-Width) DIV 2;   { Negative X centers box }
  WITH GrafChars DO
    BEGIN                                { Draw top line }
      GotoXY(X,Y); Write(ULCorner);
      FOR I := 3 TO Width DO Write(HBar);
      Write(URCorner);
                                         { Draw bottom line }
      GotoXY(X,(Y+Height)-1); Write(LLCorner);
      FOR I := 3 TO Width DO Write(HBar);
      Write(LRCorner);
                                         { Draw sides }
      FOR I := 1 TO Height-2 DO
        BEGIN
          GotoXY(X,Y+I); Write(VBar);
          GotoXY((X+Width)-1,Y+I); Write(VBar)
        END
    END
END;



BEGIN
  Randomize;                   { Seed the pseudorandom number generator }
  ClrScr;                      { Clear the entire screen }
  DefineChars(GrafChars);      { Go get box-draw characters for this machine }
  WHILE NOT KeyPressed DO      { Draw boxes until a key is pressed }
    BEGIN
      X := Random(72);         { Get a Random X/Y for UL Corner of box }
      Y := Random(21);
      REPEAT Width := Random(80-72) UNTIL Width > 1;  { Get Random Height & }
      REPEAT Height := Random(25-Y) UNTIL Height > 1; { Width to fit on CRT }
      MakeBox(X,Y,Width,Height,GrafChars);            { and draw it! }
    END
END.
```

The **GrafRec** type exists to overcome the problem that even character graphics are done in widely different ways on different computers. The IBM PC has a whole series of graphics characters in its "high" 128 character set, but the Xerox 820 has only a few drawing characters, and they can only be drawn by preceding them with a "lead-in" character. Thus, 820 graphics characters are not characters at all, but strings, hence the types in the record are not characters but short strings. Some computers have no graphics characters at all. In those cases, plus symbols + can be used for the corners, cross, and T's, and the ASCII vertical bar | and dash - for vertical and horizontal lines. The important thing is that you needn't change any code to move this procedure to different computers; you need only change the values in the **GrafChars** record passed to it.

Program **BoxTest** is one of the few programs in this book that I consciously designed to be independent of a specific hardware architecture. Turbo Pascal 4.0 and 5.0 are only available for the PC architecture, but by removing one line (the **USES** statement) from **BoxTest**, you can compile the program on any computer capable of running Turbo Pascal 3.0 as a demonstration of its machine independence.

Procedure **MakeBox** has a parameter list with five parameters in it. In the parameter list of the procedure's declaration they are named: **X**, **Y**, **Width**, **Height**, and **GrafChars**. Notice that the types of these parameters are given in the procedure declaration. **X**, **Y**, **Width**, and **Height** are all identical types, so they may be given as a list separated by commas. You could also have defined each of the four separately, like this:

```
PROCEDURE MakeBox(X         : Integer;
                  Y         : Integer;
                  Width     : Integer;
                  Height    : Integer;
                  GrafChars : GrafRec);
```

The parameters defined in a procedure's declaration are called *formal parameters* and must always be given a type, separated from the formal parameter by a colon. In the example, **X**, **Y**, **Width**, **Height**, and **GrafChars** are all formal parameters.

When a procedure is invoked, values are passed to the procedure through its parameters. The parameter types are not given:

```
MakeBox(25,BoxNum+2,30,3,Chars820);
```

Furthermore, in this example, the values may be values stored in variables, or values expressed as constants or expressions. The parameters that are present in the parameter list of a particular invocation of a procedure are called *actual parameters*. (All parameters passed to **MakeBox** are passed by value. If they were passed "by reference," the actual parameters would have to be variables of identical type to the formal parameters. This will be fully explained in the next section.)

Identifiers used as formal parameters are local to their procedure or function. As such, their names may be identical to identifiers defined in other procedures or functions, or in the main program, without any conflict. You'll find more on this particular subject in Chapter 3. The **X** and **Y** formal parameters in **MakeBox** have no relation at all to an **X** or **Y** identifier used elsewhere in the program. Of course, the flipside of this is also true: If you are using an **X** or **Y** variable global to the entire program, they will *not* be accessible from within **MakeBox**. If you try to access a global **X** or **Y** variable from within **MakeBox**, you will access the formal parameters **X** and **Y** instead.

Also remember that formal **VAR** parameters may not act as control variables in **FOR** loops. Local variables, such as **I** in **MakeBox**, should be declared for this purpose.

In the program **BoxTest**, values are loaded into **GrafChars** with assignment statements, in the procedure **DefineChars**. A wiser move would be to store the **GrafChars** record in a file, and read the record from the file into memory at program startup. That way, nothing specific to the IBM PC is baked into the program itself, and the same source could be compiled under a pre-4.0 version of Turbo Pascal on any computer.

14.2: PASSING PARAMETERS BY VALUE OR BY REFERENCE

When a function or a procedure is invoked, the actual parameters are *meshed* with the formal parameters, and then the function or procedure does its work. The meshing of actual parameters with formal parameters is done two ways: by value and by reference.

Passing Parameters by Value

A parameter passed by value is just that: a value is copied from the actual parameter into the formal parameter. The movement of the value into the procedure is a one-way street. Nothing can come back out again to be used by the calling program. This applies whether the actual parameter is a constant, an expression, or a variable.

There are powerful advantages to one-way data movement *into* a procedure. The procedure can fold, spindle, and mutilate the parameter any way it needs to, and not fear any side effects outside of the procedure. The copy of the actual parameter it gets is a truly private copy, strictly local to the procedure itself.

If a variable is passed to a procedure by value the type of the variable must be compatible with the type of the formal parameter.

Passing Parameters by Reference

There are many occasions when the whole point of passing a parameter to a function or procedure is to have it modified and returned for further use. To have a procedure or function modify a parameter and return it, the parameter must be passed by reference.

Unlike parameters passed by value, a parameter passed by reference (often called a **VAR** parameter) cannot be a literal, a constant, or an expression. By definition, the values of constants and literals cannot be changed, and the notion of changing the value of an expression and stuffing it back into the expression makes no logical sense.

To be passed by reference, an actual parameter must be a variable of the *identical* type as the formal parameter. Compatible types will not do; the types must evaluate down to the same type definition statement, as explained in Section 12.2.

The one exception to this rule in Turbo Pascal involves string types. Strings, if you recall from Section 9.5, may be defined in any physical length from 1 to 255, with the default length being 255 unless you specify some other length. Under strict type checking a **VAR** string parameter passed to a procedure must be of the identical type declared in the procedure's header:

```
VAR
  String1 : String80;
  String2 : String30;

PROCEDURE Grimble(VAR WorkString : String255);
```

In this example, strict type checking would prohibit passing either **String1** or **String2** as a parameter to procedure **Grimble**.

However, strict type checking may be relaxed with the **$V** compiler directive (see Section 27.3) to allow strings of any physical length to be passed as **VAR** parameters regardless of the formal **VAR** parameter's physical length. To relax strict type checking include this directive in your code:

```
{$V-}
```

Note that the default for type checking is strict, and you must explicitly use the **$V-** directive to relax type checking if desired.

The draconian nature of strict type checking for **VAR** parameters makes a little more sense when you realize that the variable itself is not copied into the formal parameter (as with parameters passed by value). What is passed is actually a pointer to the variable itself. Data are not being moved from one variable to another. Data are being read from one variable and written back into the same variable. To protect other data items that may exist to either side of the variable passed by reference, the compiler insists on a *perfect* match between formal and actual parameters.

The procedure **MakeBox** had several parameters, all passed by value. For an example of a parameter passed by reference, consider the Shell sort procedure below:

```
{->>>>ShellSort<<<<--------------------------------------------}
{                                                               }
{ Filename : SHELSORT.SRC -- Last Modified 7/14/88              }
{                                                               }
{ This is your textbook Shell sort on an array of key records, }
{ defined as the type show below:                               }
{                                                               }
{     KeyRec = RECORD                                           }
{                Ref     : Integer;                             }
{                KeyData : String30                             }
{              END;                                             }
{                                                               }
{                                                               }
{                                                               }
{---------------------------------------------------------------}

PROCEDURE ShellSort(VAR SortBuf : KeyArray; Recs : Integer);

VAR
  I,J,K,L : Integer;
  Spread  : Integer;


PROCEDURE KeySwap(VAR RR,SS : KeyRec);

VAR
  T : KeyRec;

BEGIN
  T := RR;
  RR := SS;
  SS := T
END;


BEGIN
  Spread := Recs DIV 2;          { First Spread is half record count  }
  WHILE Spread > 0 DO            { Do until Spread goes to zero:      }
    BEGIN
      FOR I := Spread + 1 TO Recs DO
        BEGIN
          J := I - Spread;
          WHILE J > 0 DO
            BEGIN                { Test & swap across the array }
              L := J + Spread;
              IF SortBuf[J].KeyData <= SortBuf[L].KeyData THEN J := 0 ELSE
                KeySwap(SortBuf[J],SortBuf[L]);
              J := J - Spread
            END
        END;
      Spread := Spread DIV 2    { Halve Spread for next pass }
    END
END;
```

This procedure sorts an array of sort keys. A sort key is a record type that consists of a piece of data and a pointer to a file entry from which the data came. The fastest and safest way to sort a file is not to sort the file at all, but to build an array of sort keys from information in the file and sort the array of sort keys instead.

The array then can be written out to a file. Because the data in the array are in sorted order (usually alphabetical), the array can be searched using a fast binary search function. Once a match to a desired string is found in the **Key** field of a **KeyRec** record, the **RecNum** field contains the physical record number of the record in the file where the rest of the information is stored.

Look at the parameter line for **ShellSort**:

```
PROCEDURE ShellSort(VAR SortBuf : KeyArray; Recs : Integer);
```

The first parameter, **SortBuf**, is passed by reference. The second parameter, **Recs**, is passed by value. The difference is that **SortBuf** is preceded by the keyword **VAR**. Keyword **VAR** indicates that the parameter following it is passed by reference.

The reason for passing **SortBuf** by reference should be obvious: We want to rearrange the sort keys in **SortBuf** and put them in a certain order. **ShellSort** does this rearranging. We will need to get **SortBuf** "back" when the rearranging is done. Had we passed **SortBuf** to **ShellSort** by value, **ShellSort** would have received its own private copy of **SortBuf**, would have sorted the copy, and then would have had no way to return the sorted copy to the rest of the program.

Recs contains a count of the number of sort keys loaded into the array **SortBuf**. While knowing the value stored in **Recs** is essential to sorting **SortBuf** correctly, it need not be changed, and thus **Recs** can be passed by value. Only the *value* of **Recs** is needed.

To sum up: An actual parameter passed by value is copied into the formal parameter. The copy is local to the procedure or function and changes made to the copy do not "leak out" into the rest of the program.

Passing a parameter to a procedure by reference actually gives the procedure a pointer to the physical variable being passed. Changes made to the parameter within the procedure are actually made to the physical variable outside the procedure.

To pass a parameter by reference, precede the parameter by the keyword **VAR**. When passed by reference, actual parameters must be variables of *identical type* to the formal parameter.

14.3: RECURSION

Recursion is one of those peculiar concepts that seem to defy total understanding. It depends completely on mystery for its operation, until eventually some small spark of understanding happens, and then, *wham!* It becomes simple or even obvious. A great many people have trouble understanding recursion at first glance, and if you do too, don't think less of yourself for it. For the beginner, recursion is simple. But it is *not* obvious.

Recursion is what we call the event in which a function or procedure invokes itself. It seems somehow intuitive to beginners that having a procedure call itself is either impossible or else an invitation to disaster. Both of these fears are unfounded, of course. Let's look at both of them.

Recursion is indeed possible. In fact, having a procedure call itself is no different, from a coding perspective, from having a procedure call any other procedure. What happens when a procedure calls another procedure? First, the called procedure is *instantiated;* that is, its formal parameters and local variables are allocated on the system stack. Next, the return address (i.e., the location in the code from which the procedure was called and to which it must return control) is "pushed" onto the system stack. Finally, control is passed to the called procedure's code.

When the called procedure has executed, it retrieves the return address from the system stack and then clears its variables and formal parameters off the stack by a process we call "popping." Then, it returns control to the code that called it by branching to the return address.

None of this changes when a procedure calls itself. Upon a recursive call to itself, new copies of the procedure's formal parameters and local variables are instantiated on the stack. Then control is passed to the start of the procedure again.

The problem of understanding recursion shows up when execution reaches the point in the procedure where it calls itself. A third instance of the procedure is allocated on the stack, and the procedure begins running again. There is a fourth instance, a fifth, and after a few hundred recursive calls the stack has grown so large that it collides with something important in memory, and the system crashes. If you had this kind of procedure, such a thing would happen very quickly:

```
PROCEDURE Fatal;

BEGIN
  Fatal
END;
```

This situation is an unlimited feedback loop. It is this possibility that makes newcomers feel uneasy about recursion.

Obviously, the important part of recursion is knowing when to stop.

A recursive procedure must test some condition before it calls itself. This is to see if still needs to call itself to complete its work. This condition could be a comparison of a counter against a predetermined number of recursive calls or some Boolean condition that becomes true or false when the time is right to stop recursing and go home.

When controlled in this way, recursion becomes a very powerful and elegant way to solve certain programming problems.

Let's go through a simple example of a controlled recursive procedure. Read through this code *very* carefully:

```
PROGRAM PushPop;

CONST
  Levels = 5;

VAR
  Depth : Integer;

PROCEDURE Dive(VAR Depth : Integer);

BEGIN
  Writeln('Push!');
  Writeln('Our depth is now: ',Depth);
  Depth := Depth +1;
  IF Depth <= Levels THEN Dive(Depth);
  Writeln('Pop!')
END;


BEGIN
  Depth := 1;
  Dive(Depth);
END.
```

Actually, the program is doing nothing more than setting a counter to 1 and calling the recursive procedure **Dive**. Note constant **Levels**. **Dive** prints the word "Push!" when it begins executing and the word "Pop!" when it ceases executing. In between, it prints the value of the variable **Depth** and then increments it.

If, at this point, the value of **Depth** is less than the constant **Levels**, **Dive** calls itself. Each call to **Dive** increments **Depth** by 1, until, at last, **Depth** is greater than **Levels**. Then recursion stops.

Running program **PushPop** produces this output. Can you explain to yourself exactly why?

```
Push!
Our depth is now 1
Push!
Our depth is now 2
Push!
```

```
Our depth is now 3
Push!
Our depth is now 4
Push!
Our depth is now 5
Pop!
Pop!
Pop!
Pop!
Pop!
```

Follow the execution of **PushPop** through, with a pencil to touch each keyword, if necessary, until the output makes sense to you.

14.4: APPLICATIONS OF RECURSION

Certain programming problems simply cry out for recursive solutions. Perhaps the simplest and best-known is the matter of calculating factorials. A *factorial* is the product of a digit and all the digits less than it, down to one:

```
5! = 5 * 4 * 3 * 2 * 1
```

A little scrutiny here will show that 5! is the same as 5 * 4!, and 4! is the same as 4 * 3!, and so on. In the general case, N! = N * (N-1)! Whether you see it immediately or not, we have already expressed the factorial algorithm recursively by defining it in terms of a factorial. This will become a little clearer when we express it in Pascal:

```
FUNCTION Factorial(N : Integer) : Integer;

BEGIN
  IF N > 1 THEN Factorial := N * Factorial(N-1)
    ELSE Factorial := 1
END;
```

And that is it. We express it as a conditional statement because there must always be something to tell the code when to stop recursing. Without the N > 1 test the function would merrily decrement N down past zero and recurse away until the system crashed.

The way to understand this function is to work it out for N=1, then N=2, then N=3, and so on. For N=1, the N > 1 test returns **FALSE**, so is assigned the value 1. There is no recursion involved. 1! = 1. For N=2, a recursive call to **Factorial** is made: **Factorial** is assigned the value **2 * Factorial(1)**. As we saw above, **Factorial(1) = 1**. So 2! = 2 * 1, or 2. For N=3, two recursive calls are made: **Factorial** is assigned the value **3 * Factorial(2)**. **Factorial(2)** is computed (as we just saw) by evaluating (recursively)

2 * Factorial(1). And **Factorial(1)** is simply = 1. Are you catching on? An interesting thing to do is to add temporarily a **Writeln** statement to **Factorial** that displays the value of **N** at the beginning of each invocation.

A sidenote on the power of factorials: Calculating anything over 7! will overflow a 2-byte integer.

A Recursive Quicksort Procedure

A considerably more useful application of recursion lies in the *quicksort* method of sorting arrays, invented by noted computer scientist C.A.R. Hoare. Quicksort procedures can be written in a number of different ways, but the simplest way is by using recursion.

This will not be an easy procedure to understand if you are a beginner. If you can't make sense of it right now, come back to it after you have had a chance to use Pascal for awhile.

The quicksort procedure below does the same job that the procedure **ShellSort** did in the last section. **QuickSort** is passed an array of **KeyRec** and a count of the number of records to be sorted in the array. It rearranges the records until they are in ascending sort order in the array:

```
{->>>>QuickSort<<<<------------------------------------------}
{                                                            }
{ Filename : QUIKSORT.SRC -- Last Modified 7/14/88           }
{                                                            }
{ This is your textbook recursive quicksort on an array of key }
{ records, which are defined as the type show below:         }
{                                                            }
{     KeyRec = RECORD                                        }
{                Ref     : Integer;                          }
{                KeyData : String30                          }
{              END;                                          }
{                                                            }
{                                                            }
{                                                            }
{------------------------------------------------------------}

PROCEDURE QuickSort(VAR SortBuf : KeyARRAY;
                        Recs    : Integer);


PROCEDURE KeySwap(VAR RR,SS : KeyRec);

VAR
  T : KeyRec;

BEGIN
  T := RR;
  RR := SS;
  SS := T
END;


```

```
PROCEDURE DoSort(Low, High : Integer);

VAR
  I,J   : Integer;
  Pivot : KeyRec;

BEGIN
  { Can't sort if Low is greater than or equal to High... }
  IF Low < High THEN
    BEGIN
      I := Low;
      J := High;
      Pivot := SortBuf[J];
      REPEAT
        WHILE (I < J) AND (SortBuf[I].KeyData <= Pivot.KeyData) DO I := I + 1;
        WHILE (J > I) AND (SortBuf[J].KeyData >= Pivot.KeyData) DO J := J - 1;
        IF I < J THEN KeySwap(SortBuf[I],SortBuf[J]);
      UNTIL I >= J;
      KeySwap(SortBuf[I],SortBuf[High]);
      IF (I - Low < High - I) THEN
        BEGIN
          DoSort(Low,I-1);    { Recursive calls to DoSort! }
          DoSort(I+1,High)
        END
      ELSE
        BEGIN
          DoSort(I+1,High);    { Recursive calls to DoSort! }
          DoSort(Low,I-1)
        END
    END
END;


BEGIN
  DoSort(1,Recs);
END;  { QuickSort }
```

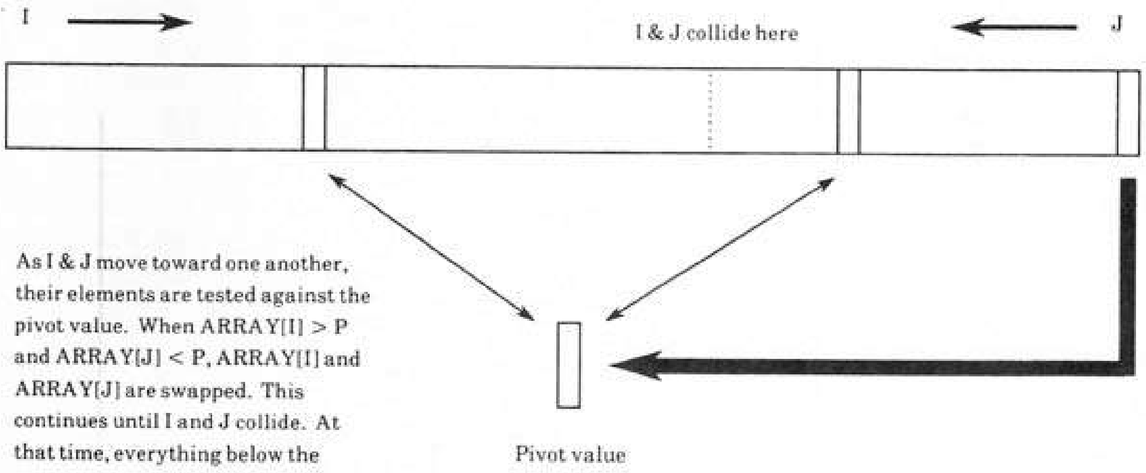
QuickSort's modus operandi is summarized in Figure 14.1. One of the elements is chosen arbitrarily to be the *pivot value*. Here, it is the last element in the array. The idea is to divide the array into two partitions such that all elements on one side of the partition are greater than the pivot value, and all elements on the other side of the partition are less than the pivot value.

This is done by scanning the array from both ends toward the middle by counters **I** and **J**. **I** scans from the low end upward; **J** from the high end downward. The **I** counter samples each element, and stops when it finds an element whose value is *higher* than the pivot value. Then the scan begins from the top end down, with the **J** counter looking for a value that is *less* than the pivot value. When it finds one, the two found elements are swapped, thus putting them on the proper side of the pivot value.

When **I** and **J** collide in the middle somewhere (*not* necessarily in the center!), the array has been partitioned into two groups of elements: One that is larger than the pivot value, and one that is smaller than the pivot value. These two groups are not

Figure 14.1

A Quicksort Scan



As I & J move toward one another, their elements are tested against the pivot value. When ARRAY[I] > P and ARRAY[J] < P, ARRAY[I] and ARRAY[J] are swapped. This continues until I and J collide. At that time, everything below the collision point < P, and everything above the collision point > P. The two groups are thus sorted with respect to one another

necessarily equal in size, and usually won't be. The only thing certain is that all the elements in one group are less than the value of the pivot element, and all the elements in the other group are greater than the pivot element. The two *groups* are sorted with respect to one another: All elements of the low group are less than all elements of the high group.

Enter recursion: This same process is now applied to each of the two groups by calling **DoSort** recursively for each group. A new pivot value is chosen for each group, and each group is partitioned around its pivot value, just as the entire array was originally. When this is done there are four groups. A little thought will show you that low-valued elements of the array are being driven toward the low end of the array, and high-valued elements are being driven toward the high end of the array. Within each group there is no guarantee that the elements are in sorted order. What you must understand is that the *groups themselves* are in sort order. In other words, *all* the elements of one group are greater than all the elements of the group below it.

To press on: Each of the four groups is partitioned again by more recursive calls to **DoSort**. The groups are smaller. Each group taken as one is sorted with respect to all other groups. With each recursive call, the groups have fewer and fewer members. In time, each group will contain only one element. Given that the groups are always in sort order, if each group is a single element, then all elements of the array are in sorted order, and **QuickSort**'s job is finished.

How does **QuickSort** know when to stop recursing? The first conditional test in **DoSort** does it: If **Low** is greater than or equal to **High**, the sort is finished. Why? Because **Low** and **High** are the bounds of the group being partitioned. If **Low** = **High**, the group has only one member. When the groups have only one member, the array is in sort order and work is done.

If this makes your head spin, you are in good company. Follow it through a few times until it makes sense. Once you can follow **QuickSort**'s internal logic, you will have a *very* good grasp of the uses of recursion.

This particular Quicksort algorithm works best when the original order of the elements in the array is random or nearly so. It works least well when the original order is close to fully sorted. For an array of random elements, it is one of the fastest of all sorting methods. For sorting arrays that are close to being in order, the **ShellSort** procedure given earlier will be consistently faster.

The following programs puts the two sort procedures to the test. It also will give you a taste of using keyed files. It generates a file of random keys, allows you to display the random keys to verify how random they are. (Thoroughly.) Finally, it will sort the file by either of the two methods. Once the file has been sorted, you can display the keys once more to be sure that they have in fact been sorted.

```
{--------------------------------------------------------------}
{                          SortTest                            }
{                                                              }
{               Data sort demonstration program                }
{                                                              }
{                             by Jeff Duntemann                }
{                             Turbo Pascal V5.0                }
{                             Last update 7/14/88              }
{                                                              }
{                                                              }
{                                                              }
{--------------------------------------------------------------}

PROGRAM Sorttest;

USES CRT,DOS,BoxStuff;

CONST
  HighLite    = True;
  CR          = True;
  NoHighlite  = False;
  NoCR        = False;
  GetInteger  = False;
  Numeric     = True;
  CapsLock    = True;
  Shell       = True;
  Quick       = False;


 TYPE
   String255 = String[255];
   String80  = String[80];
   String30  = String[30];

  KeyRec = RECORD
             Ref     : Integer;
             KeyData : String30
           END;

  KeyArray = ARRAY[0..500] OF KeyRec;

  KeyFile = FILE OF KeyRec;


VAR
  I,J,Error : Integer;
  IVAL      : Integer;
  R         : Real;
  Ch        : Char;
  Response  : String80;
  Escape    : Boolean;
  WorkArray : KeyArray;
  Randoms   : KeyFile;


{$I BEEP.SRC}       { Described in Section 16.13 }
{$I UHUH.SRC}       { Described in Section 16.13 }
{$I PULL.SRC }      { Described in Section 16.12 }
{$I CLREGION.SRC}   { Described in Section 18.1 }
{$I CURSON.SRC}     { Described in Section 18.4 }
{$I CURSOFF.SRC}    { Described in Section 18.4 }
{$I YES.SRC }       { Described in Section 18.3 }
{$I WRITEAT.SRC}    { Described in Section 18.3 }
{$I GETSTRIN.SRC}   { Described in Section 15.2 }
{$I SHELSORT.SRC}   { Described in Section 14.2 }
{$I QUIKSORT.SRC}   { Described in Section 14.4 }



PROCEDURE GenerateRandomKeyFile(KeyQuantity : Integer);

VAR WorkKey : KeyRec;
    I,J     : Integer;

BEGIN
  Assign(Randoms,'RANDOMS.KEY');
  Rewrite(Randoms);
  FOR I := 1 TO KeyQuantity DO
    BEGIN
      FillChar(WorkKey,SizeOf(WorkKey),0);
      FOR J := 1 TO SizeOf(WorkKey.KeyData)-1 DO
        WorkKey.KeyData[J] := Chr(Pull(65,91));
      WorkKey.KeyData[0] := Chr(30);
      Write(Randoms,WorkKey);
    END;
  Close(Randoms)
END;


PROCEDURE DisplayKeys;

```

```
VAR WorkKey : KeyRec;

BEGIN
  Assign(Randoms,'RANDOMS.KEY');
  Reset(Randoms);
  Window(25,13,70,22);
  GotoXY(1,1);
  WHILE NOT EOF(Randoms) DO
    BEGIN
      Read(Randoms,WorkKey);
      IF NOT EOF(Randoms) THEN Writeln(WorkKey.KeyData)
    END;
  Close(Randoms);
  Writeln;
  Writeln('        >>Press (CR)<<');
  Readln;
  ClrScr;
  Window(1,1,80,25)
END;



PROCEDURE DoSort(Shell : Boolean);

VAR Counter : Integer;

BEGIN
  Assign(Randoms,'RANDOMS.KEY');
  Reset(Randoms);
  Counter := 1;
  WriteAt(20,15,NoHighlite,NoCR,'Loading...');
  WHILE NOT EOF(Randoms) DO
    BEGIN
      Read(Randoms,WorkArray[Counter]);
      Counter := Succ(Counter)
    END;
  Close(Randoms);
  Write('...sorting...');
  IF Shell THEN ShellSort(WorkArray,Counter-1)
    ELSE QuickSort(WorkArray,Counter-1);
  Write('...writing...');
  Rewrite(Randoms);
  FOR I := 1 TO Counter-1 DO Write(Randoms,WorkArray[I]);
  Close(Randoms);
  Writeln('...done!');
  WriteAt(-1,21,NoHighlite,NoCR,'>>Press (CR)<<');
  Readln;
  ClearRegion(2,15,77,22)
END;



BEGIN
  ClrScr;
  CursorOff;
  MakeBox(1,1,80,24,GrafChars);
  WriteAt(24,3,HighLite,NoCR,'THE COMPLETE TURBO PASCAL SORT DEMO');
  REPEAT
```

```
      WriteAt(25,5,NoHighlite,NoCR,'[1] Generate file of random keys');
      WriteAt(25,6,NoHighlite,NoCR,'[2] Display file of random keys');
      WriteAt(25,7,NoHighlite,NoCR,'[3] Sort file via Shell sort');
      WriteAt(25,8,NoHighlite,NoCR,'[4] Sort file via Quicksort');
      WriteAt(30,10,NoHighlite,NoCR,'Enter 1-4: ');
      Response := ''; IVal := 0;
      GetString(46,10,Response,2,CapsLock,Numeric,GetInteger,
                R,IVal,Error,Escape);
      CASE IVal OF
        0 :;
        1 : GenerateRandomKeyFile(250);
        2 : DisplayKeys;
        3 : DoSort(Shell);
        4 : DoSort(Quick);
        ELSE
      END; {CASE}
    UNTIL (IVal = 0) OR Escape;
    CursorOn
  END.
```

14.5: FORWARD DECLARATIONS

In discussing pointer variables and how to declare them back in Section 10.2, I pointed out a departure from Pascal practice: You can use an identifier in a pointer definition before that identifier is defined. In other words, this pair of type definitions is completely legal:

```
TYPE
  RecPtr = ^DynaRec;
  DynaRec = RECORD
              DataPart : String;
              Next     : RecPtr
            END;
```

Turbo Pascal "takes our word" that, in fact, we will define **DynaRec** before the program ends, and allows the definition of **RecPtr**. This use of an undefined identifier is called a "forward reference."

The context of defining pointer types is the *only* context in which Pascal will accept a forward reference without any special persuasion. In certain circumstances, however, Pascal can be persuaded to accept a procedure or function identifier before that procedure or function has been defined. In a sense, we must promise the compiler that, in fact, we will define the identifier. Or, if you prefer, we have to declare that we will declare such an identifier somewhere down the source code trail.

This promise is called a "forward declaration." It is accomplished with a Pascal reserved word, **FORWARD**, and it is done this way:

```
PROCEDURE NotThereYet(Foo,Bar : Integer); FORWARD;
```

What we have here is the procedure header all by itself, without any procedure body or local declarations of constants, types, or variables. People who understand Turbo Pascal units (see Chapter 17) will think that this resembles declarations in the interface section of a unit, and they will be right.

Later on in the program, sometime before the **BEGIN** that marks the start of the main program block, procedure **NotThereYet** must be declared in its entirety. If it isn't, Turbo Pascal will give us this error:

```
Error 59: Undefined forward (NOTHEREYET)
```

The eventual declaration is perfectly ordinary. No special syntax indicates that the procedure had earlier been declared as **FORWARD**.

You do have the option of *not* redeclaring the forward-declared procedure's parameter list. In other words, you could in fact define an empty procedure **NotThereYet** without parameters:

```
PROCEDURE NotThereYet;

BEGIN
END;
```

This also will be familiar to people who understand Turbo Pascal's units. I consider it bad practice, however. The compiler allows you to redeclare the parameters, and it contributes to the clarity of the program to have the parameter list in both places: the forward declaration (in which they are essential) and the full definition.

At last we come to the question: What good is all of this? The answer is: not much. The only situation that genuinely requires forward declaration is circular (also called "mutual") recursion. Consider these two procedure definitions:

```
PROCEDURE Egg;

BEGIN
  .
  .
  Chicken;
  .
  .
END;


PROCEDURE Chicken;

BEGIN
  .
  .
  Egg;
  .
  .
END;
```

Which comes first? You can't declare **Chicken** without calling **Egg**, and you can't declare **Egg** without calling **Chicken**. Pascal will call foul on the whole thing unless you forward-declare one or the other. Adding the forward declaration makes everything copacetic:

```
PROCEDURE Chicken; FORWARD;

PROCEDURE Egg;

BEGIN
  .          { Other program logic keeps this from }
  .          { being an infinite loop. }
  Chicken;   { Without the FORWARD, we get an error here. }
  .
  .
END;


PROCEDURE Chicken;

BEGIN
  .
  .
  Egg;
  .
  .
END;
```

That is what circular recursion *is*, but what it's good for thus far has escaped me. If any reader has come across a use more compelling than the mathematical oddity **Catch22** in the old *Turbo Pascal 3.0 Reference Manual*, I would like to hear about it. My hunch is that any program that appears to require circular recursion can probably be rewritten a different way without it.

I look upon the reserved word **FORWARD** much as I do the spokeshave sitting in the bottom drawer of my tool cabinet. I have never used it, but, by God, if I ever need to shave some spokes, I know just where it is.

15 How to Use Strings

ISO Standard Pascal lacks a "clean" way to deal with text. There is no **String** type in ISO Pascal. To work with strings in ISO Pascal, you must confine your text to arrays of characters, keep your own logical length counters, and write all the procedures for manipulating strings yourself.

Given the pervasiveness of text in the work that computers do, it is not surprising that most implementors of Pascal have extended the language by providing a **String** data type and some built-in procedures and functions to manipulate strings.

Turbo Pascal is no exception. We looked briefly at the **String** data type in Section 9.5. A **String** variable is actually an array of characters with a counter attached to keep tabs on how many characters have been loaded into the array. The Turbo Pascal type **String** has a physical length of 255 characters.

You can define physically shorter string types by using the word **String** followed by the maximum physical length of the string (up to 255) in brackets. For example, a string type capable of holding 80 characters of information would be defined this way:

```
TYPE
  String80 = String[80];
```

The type **String80** is actually 81 physical bytes long. Byte 0 is the counter. It can hold a number from 0 to 80. Bytes 1 through 80 hold the actual text characters that make up the string. It may be helpful to think of this string type as a record type:

```
TYPE
  String80 = RECORD
               Counter : Char;
               ARRAY[1..80] OF Char
             END;
```

This is just a comparison, however. Type **String** is treated specially by the compiler in a number of ways, and you cannot define a record type named **String** and expect it to get the same special treatment.

When you assign a string literal to a string variable, the length counter is automatically updated to reflect the number of characters assigned:

```
VAR
  MyText : String;

MyText := '';
MyText := 'Let the wookie win.';
```

Before the text assignment, **MyText** had been assigned the null string '' and its length counter was 0. After the assignment, the length counter was automatically updated to 19.

Turbo Pascal provides a number of powerful string procedures and functions for manipulating text stored in strings. In the next section we'll look at each in detail.

15.1: BUILT-IN STRING PROCEDURES AND FUNCTIONS

Length

The length counter of a string variable is accessible two ways. One way is simply to examine element 0 of the string:

```
Count := Ord(MyText[0]);
```

This will put the length counter of string **MyText** into the integer variable **Count**. The length counter of string variables, just like the data in the string itself, is type **Char**, which needs the transfer function **Ord** to be assigned to an integer.

A considerably better way to do the same thing is to use the predefined function **Length**. It is a built-in function predefined this way:

```
FUNCTION Length(Target : String) : Integer;
```

Length returns the value of the length counter byte, which indicates the logical length of string **Target**:

```
Count := Length(MyText);
```

This is functionally equivalent to accessing the length byte directly and it is somewhat easier to read. **Length** is a function that returns an integer value.

The length of the null string ('') is 0.

The following procedure **CapsLock** is a variation of a procedure originally presented in Chapter 9 in connection with sets. **CapsLock** accepts a string parameter and returns it with all lower case letters changed to their corresponding upper case letters. Note the use of the **Length** function:

```
PROCEDURE CapsLock(VAR MyString : String);

VAR
  I : Integer;

BEGIN
  FOR I := 1 TO Length(MyString) DO
    MyString[I] := UpCase(MyString[I]);
END;
```

This **CapsLock** makes use of a convenient Turbo Pascal built-in function, **UpCase**. **UpCase** accepts a character value as a parameter and returns the uppercase equivalent of that character if the parameter is lowercase. If the parameter is not a lowercase character it is returned unchanged.

Keep in mind that to pass strings physically shorter than 255 characters to **CapsLock** (or any subprogram with a string VAR parameter), you need to relax strict type checking with the **{$V-}** compiler command.

Length as a "Reversible" Function

Under Turbo Pascal 4.0 only, the **Length** function has a truly remarkable property: It can be placed on the *other* side of an assignment operator, and *set* the length of a string rather than read the length of a string.

```
Length(MyString) := 22;
```

This example will force the length counter of **MyString** to a value of 22, regardless of the contents of the string. Any characters in the string beyond number 22 will be effectively truncated.

Use some caution when working with **Length** this way. Pushing the length counter *beyond* the logical length of the string can have the effect of adding garbage characters to the string, some of which may not be printable ASCII.

Concat

Concatenation is the process of combining two or more strings to form a single string. Turbo Pascal gives you two separate ways to perform this operation.

The easiest way to concatenate two or more strings is to use Turbo Pascal's string concatenation operator +. Many BASIC interpreters also use the plus symbol to concatenate strings. Simply place the string variables in order, separated by the string concatenation operator:

```
BigString := String1 + String2 + String3 + String4;
```

String variable **BigString** should, of course, be large enough to hold all the variables you intend to concatenate into it. If the total length of all the source strings is greater than the physical length of the destination string, all data that will not fit into the destination string is truncated and ignored.

The built-in function **Concat** performs the same function as the string concatenation operator. It is included in Turbo Pascal because several older Pascal compilers (notably UCSD Pascal and Pascal/MT+) use a **Concat** function and do not have a string concatenation operator. **Concat** is a function returning a value of type **String**. It accepts any number of string variables and string literals as parameters, which are separated by commas.

The following example shows how both string concatenation methods work:

```
VAR
  Subject,Predicate,Sentence : String[80];

Subject := 'Kevin the hacker';
Predicate := 'crashed the system';
Sentence  := Subject + Predicate;
Sentence :=
  Concat(Sentence,', but brought it up again.';
Writeln(Sentence);
```

Here, two string variables and a string literal are concatendated into a single string variable. The CRT output of the **Writeln** statement would be the concatenated string:

```
Kevin the hacker crashed the system, but brought it up again.
```

Delete

Removing one or more characters from a string is the job done by the built-in **Delete** procedure, which is predefined this way:

```
PROCEDURE Delete(Target : String; Pos,Num : Integer);
```

Delete removes **Num** characters from the string **Target** beginning at character number **Pos**. The length counter of **Target** is updated to reflect the deleted characters.

```
VAR
  Magic : String;

Magic := 'Watch me make an elephant disappear...';
Delete(Magic,15,11);
Writeln(Magic);
```

Before the **Delete** operation, the string **Magic** has a length of 38 characters. When run, this example will display:

```
Watch me make  disappear...
```

The new length of **Magic** is set to 27.

One use of **Delete** is to remove *leading whitespace* from a string variable. Whitespace is a set of characters that includes space, tab, carriage return, and linefeed. Whitespace is used to format text files for readability by human beings. However, when that text file is read by computer, the whitespace must be removed, as it tells the computer nothing.

The following procedure strips leading whitespace from a string variable:

```
PROCEDURE StripWhite(VAR Target : String);

CONST
  Whitespace : SET OF Char = [#8,#10,#12,#13,' '];

BEGIN
  WHILE (Length(Target) > 0) AND (Target[1] IN Whitespace) DO
    Delete(Target,1,1)
END;
```

Whitespace is a set constant (see Section 9.6) containing the whitespace characters. Set constants are a feature unique to Turbo Pascal. Keep that in mind if you intend to export your code to another Pascal compiler.

Delete(Target,1,1) deletes one character from the beginning of string **Target**. The second character is then moved up to take its place. If it, too, is a whitespace character it is also deleted, and so on until a non-whitespace character becomes the first character in **Target**, or until **Target** is emptied completely of characters.

Pos

Locating a substring within a larger string is handled by the built-in function **Pos**. **Pos** is predefined this way:

```
FUNCTION
  Pos(Pattern : <string or char>; Source : String) : Integer;
```

Pos returns an integer that is the location of the first occurrence of **Pattern** in **Source**. **Pattern** may be a string variable, a **Char** variable, or a string or character literal.

For example:

```
VAR
  ChX,ChY     : Char;
  Little,Big : String;

Big := 'I am an American, Chicago-born.  Chicago, that somber city.'
Little := 'Chicago';
ChX := 'g';
ChY := 'G';

Writeln('The position of ',Little,' in "Big" is ',Pos(Little,Big));
Writeln('The position of ',ChX,' in "Big" is ',Pos(ChX,Big));
Writeln('The position of ',ChY,' in "Big" is ',Pos(ChY,Big));
Writeln('The position of somber in "Big" is ',Pos('somber',Big));
Writeln('The position of r in "Big" is ',Pos('r',Big));
```

When executed, this example code will display the following:

```
The position of Chicago in "Big" is 19
The position of g in "Big" is 24
The position of G in "Big" is 0
The position of somber in "Big" is 48
The position of r in "Big" is 12
```

Pos does distinguish between upper and lower case letters. Note that if **Pos** cannot locate **Pattern** in **Source**, it returns a value of 0.

Copy

Extracting a substring from within a string is accomplished with the **Copy** built-in function. **Copy** is predefined this way:

```
FUNCTION Copy(Source : String; Index,Size : Integer) : String;
```

Copy returns a string that contains **Size** characters from **Source**, beginning at character #**Index** within **Source**:

```
VAR
  Roland,Tower : String;

Roland := 'Childe Roland to the Dark Tower came!';
Tower := Copy(Roland,22,10);
Writeln(Tower);
```

When run, this example will print:

```
Dark Tower
```

In this example, **Index** and **Size** are passed to **Copy** as constants. They can also be passed as integer variables or expressions. The following function accepts a string containing a file name, and returns a string value containing the file extension. The extension is the part of a file name from the period to the end; in "TURBO.COM" the extension is ".COM".

```
FUNCTION GetExt(FileName : String) : String;

VAR
  DotPos : Integer;
```

```
BEGIN
  DotPos := Pos('.',FileName);
  IF DotPos = 0 THEN GetExt := '' ELSE
    GetExt :=
Copy(FileName,DotPos,(Length(FileName)-DotPos)+1);
END;
```

GetExt first tests to see if there is, in fact, a period in **FileName** at all (file extensions are optional). If there is no period, there is no extension, and **GetExt** is assigned the null string. If a period is there, **Copy** is used to assign to **GetExt** all characters from the period to the end of the string.

Since the length of a file extension may be 2, 3, or 4 characters, the expression **(Length(FileName)-DotPos)+1** is needed to calculate just how long the extension is in each particular case.

You should note that, unlike most Pascal compilers (such as Pascal/MT+ and UCSD Pascal), Turbo Pascal allows you to write functions that return string values. Turbo Pascal is the only microcomputer Pascal compiler I know of that allows this. You should keep it in mind if portability is an important consideration to you.

If **Index** plus **Size** is greater than the logical length of **Source**, **Copy** truncates the returned string value to whatever characters lie between **Index** and the end of the string.

Insert

A string can be added to the end of another string by using the **Concat** function. Copying a string into the middle of another string (and not simply tacking it on at the end) is done with the **Insert** procedure.

Insert is predefined this way:

```
PROCEDURE Insert(Source     : String;
                 VAR Target : String;
                 Pos        : Integer);
```

When invoked, **Insert** copies **Source** into **Target** starting at position **Pos** within **Target**. All characters in **Target** starting at position **Pos** are moved forward to make room for the inserted string, and **Target**'s length counter is updated to reflect the addition of the inserted characters.

```
VAR
  Sentence,Ozzie : String;

Sentence := 'I am King of Kings.';
Ozzie := 'Ozymandias, ';
Insert(Ozzie,Sentence,6);
Writeln(Sentence);
```

The output from this example would be:

```
I am Ozymandias, King of Kings.
```

If inserting text into **Target** gives **Target** more characters than it can physically contain, **Target** is truncated to its maximum physical length:

```
VAR
  Fickle,GOP : String[18];

Fickle := 'I am a Democrat.';
GOP := 'Republican.';
Insert(GOP,Fickle,8);
Writeln(Fickle);
```

This prints:

```
I am a Republican.
```

Note in this example that the word "Democrat." was not overwritten; it was pushed off the end of string **Fickle** into nothingness. After the insert, **Fickle** should have contained

```
I am a Republican.Democrat.
```

however, **Fickle**, defined as **String[18]**, is only 18 physical characters long. "I am a Republican." fills it completely. "Democrat." was lost to truncation.

Str

It is important to remember (and easy enough to forget) that a number and its string equivalent are *not* interchangeable. In other words, the integer 37 and its string representation, the two ASCII characters 3 and 7 look the same on your screen but are completely incompatible in all ways but that.

Unlike most other Pascals, Turbo provides a pair of procedures for translating numeric values into their string equivalents, and vice versa.

Translating a numeric value to its string equivalent is done with the procedure **Str**. **Str** is predefined this way:

```
PROCEDURE Str(<formatted numeric value>; VAR ST : String);
```

The formatted numeric value can be either an integer or a real number. It is given as a *write parameter* (see Section 19.3 for a complete discussion of write parameters as they apply to all simple data types, numeric and non-numeric). Briefly, a write parameter is an expression that gives a numeric value and a format to express it in. The write parameter **I:7** (assuming **I** was previously declared an integer) right justifies the value of **I** in a field seven characters wide. **R:9:3** (assuming **R** was declared **Real** previously) right-justifies the value of **R** in a field 9 characters wide with three figures to the right of the decimal place.

The use of **Str** is best shown by a few examples:

```
CONST
  Bar = '|';

VAR
  R   : Real;
  I   : Integer;
  TX  : String[30];

R := 45612.338;
I := 21244;

Str(I:8,TX);
Writeln(Bar,TX,Bar);       { Displays:  |   21244| }

Str(I:3,TX);
Writeln(Bar,TX,Bar);       { Displays:  |21244|   }

Str(R,TX);
Writeln(Bar,TX,Bar);       { Displays:  |  4.5612338000E+04| }

Str(R:13:4,TX);
Writeln(Bar,TX,Bar);       { Displays:  |   45612.3380| }
```

In the third example, if you do not specify any format for a real number, the default format will be scientific notation in a field eighteen characters wide.

Val

Going in the other direction, from string representation to numeric value, is accomplished by the **Val** procedure. **Val** is predeclared this way:

```
PROCEDURE
Val(ST : String; VAR <numeric variable>; VAR Code : Integer);
```

Val's task is somewhat more complicated than **Str**'s. For every numeric value there is a string representation that may be constructed. The reverse is not true; there are many string constructions that cannot be evaluated as numbers. So **Val** must have a means of returning an error code to signal an input string that cannot be evaluated to a number. This is the purpose of the **Code** parameter.

If the string is evaluated without any problem, **Code**'s value is 0 and the numeric equivalent of the string is returned in the numeric variable. If Turbo finds that it cannot evaluate the string to a number, **Code** returns the character position of the first character that does not jive with Turbo's evaluation scheme. The numeric variable in that case is undefined:

```
PROGRAM Evaluator;

VAR
  SST    : String;
  R      : Real;
  Result : Integer;

BEGIN
  REPEAT
    Write('>>Enter a number in string form: ');
    Readln(SST);
    IF Length(SST) > 0 THEN
      BEGIN
        Val(SST,R,Result);
        IF Result <> 0 THEN
          Writeln
          ('>>Cannot evaluate that string.  Check character #',Result)
        ELSE
          Writeln
          ('>>The numeric equivalent of that string is ',R:18:10)
      END
  UNTIL Length(SST) = 0
END.
```

This little program will allow you to experiment with **Val** and see what it will accept and what it will reject. One shortcoming of **Val** is that *it considers commas an error.* A string like 5,462,445.3 will generate an error on character #2.

In the next section we will be using **Val** in slightly more sophisticated surroundings to build a generalized data entry routine.

Built-In String Routine Summary

Built-in string-handling routines

```
FUNCTION Concat(Source1,Source2...SourceN : String) : String

FUNCTION Copy(Source : String; Index,Size : Integer) : String

PROCEDURE Delete(Target : String; Index,Size : Integer)

PROCEDURE Insert(Source     : String;
                 VAR Target : String;
                 Index      : Integer)
```

```
FUNCTION Length(Source : String) : Integer

FUNCTION Pos(Pattern : String or Char;
             Source  : String) : Integer

PROCEDURE Str(Num : <write parameter>, VAR StrEquiv : String)

PROCEDURE Val(Source        : String;
              VAR NumEquiv : <Integer or Real>;
              VAR Code     : Integer)
```

15.2: A STRING INPUT PROCEDURE

Pascal provides only one built-in way to enter a string from the system console. **Readln** will accept a string from the keyboard, by waiting for characters to be typed until (CR) is pressed. As **Readln** is waiting for input, you can backspace over mistyped characters. This is a fairly typical example of entering a string from the console using **Readln**:

```
VAR
  Buff10 : String[10];

Write('Type up to 10 letters: ');
Readln(Buff10);
```

Here, once you've typed ten letters, **Readln** will allow you to type as many more as you like. However, it will only return as many characters for which the string has room. If you have set up a complicated data entry form on your CRT, nothing will stop the operator from typing beyond the right boundary of any field and disrupting other fields to the right.

What you need is a slightly more disciplined string input routine. Ideally, such a routine should show you how large the string can be, accept characters up to that limit, and then ignore further characters until previous characters are deleted with backspace, or until CR or ESC is pressed.

The following string input routine does all this and more. First, read it over and try to understand how it works:

```
{->>>>GetString<<<<-------------------------------------------}
{                                                             }
{ Filename : GETSTRIN.SRC -- Last Modified 7/14/88            }
{                                                             }
{ This is a generalized string-input procedure.  It shows a   }
{ field between vertical bar characters at X,Y, with any      }
{ string value passed initially in XString left-justified in  }
{ the field.  The current state of XString when the user      }
{ presses Return is returned in XString.  The user can press  }
{ ESC and leave the passed value of XString undisturbed, even  }
{ if XString was altered prior to his pressing ESC.            }
{                                                              }
{                                                              }
{                                                              }
{--------------------------------------------------------------}

PROCEDURE GetString(    X,Y      : Integer;
                    VAR XString  : String80;
                        MaxLen   : Integer;
                        Capslock : Boolean;
                        Numeric  : Boolean;
                        GetReal  : Boolean;
                    VAR RValue   : Real;
                    VAR IValue   : Integer;
                    VAR Error    : Integer;
                    VAR Escape   : Boolean);


VAR I,J        : Integer;
    Ch         : Char;
    Cursor     : Char;
    Dot        : Char;
    BLength    : Byte;
    ClearIt    : String80;
    Worker     : String80;
    Printables : SET OF Char;
    Lowercase  : SET OF Char;
    Numerics   : SET OF Char;
    CR         : Boolean;


BEGIN
  Printables := [' '..'}'];                { Init sets }
  Lowercase  := ['a'..'z'];
  IF GetReal THEN Numerics := ['-','.','0'..'9','E','e']
    ELSE Numerics := ['-','0'..'9'];
  Cursor := '_'; Dot := '.';
  CR := False; Escape := False;
  FillChar(ClearIt,SizeOf(ClearIt),'.');   { Fill the clear string  }
  ClearIt[0] := Chr(MaxLen);               { Set clear string to MaxLen }

                              { Convert numbers to string if required:  }
  IF Numeric THEN             { Convert zero values to null string: }
    IF (GetReal AND (RValue = 0.0)) OR
       (NOT GetReal AND (IValue = 0)) THEN XString := ''
    ELSE                      { Convert nonzero values to string equiv: }
      IF GetReal THEN Str(RValue:MaxLen,XString)
        ELSE Str(IValue:MaxLen,XString);

                                           { Truncate string value to MaxLen }
  IF Length(XString) > MaxLen THEN XString[0] := Chr(MaxLen);
  GotoXY(X,Y); Write('|',ClearIt,'|');    { Draw the field  }
  GotoXY(X+1,Y); Write(XString);
  IF Length(XString)<MaxLen THEN
    BEGIN
      GotoXY(X + Length(XString) + 1,Y);
      Write(Cursor)                                 { Draw the Cursor }
    END;
  Worker := XString;        { Fill work string with input string     }

  REPEAT                    { Until ESC or (CR) entered }
                            { Wait here for keypress:   }
    WHILE NOT KeyPressed DO BEGIN (NULL) END;
    Ch := ReadKey;

    IF Ch IN Printables THEN                  { If Ch is printable... }
      IF Length(Worker) >= MaxLen THEN UhUh ELSE
        IF Numeric AND (NOT (Ch IN Numerics)) THEN UhUh ELSE
          BEGIN
            IF Ch IN Lowercase THEN IF Capslock THEN Ch := Chr(Ord(Ch)-32);
            Worker := CONCAT(Worker,Ch);
            GotoXY(X+1,Y); Write(Worker);
            IF Length(Worker) < MaxLen THEN Write(Cursor)
          END
    ELSE  { If Ch is NOT printable... }
      CASE Ord(Ch) OF
       8,127 : IF Length(Worker) <= 0 THEN UhUh ELSE
                 BEGIN
                   Delete(Worker,Length(Worker),1);
                   GotoXY(X+1,Y); Write(Worker,Cursor);
                   IF Length(Worker) < MaxLen-1 THEN Write(Dot);
                 END;

       13 : CR := True;            { Carriage return }

       24 : BEGIN                  { CTRL-X : Blank the field }
              GotoXY(X+1,Y); Write(ClearIt);
              Worker := '';        { Blank out work string }
            END;

       27 : Escape := True;        { ESC }
       ELSE UhUh                   { CASE ELSE }
    END; { CASE }

  UNTIL CR OR Escape;              { Get keypresses until (CR) or }
                                   { ESC pressed }
  GotoXY(X + 1,Y); Write(ClearIt);
  GotoXY(X + 1,Y); Write(Worker);
  IF CR THEN                       { Don't update XString if ESC hit }
    BEGIN
      XString := Worker;
      IF Numeric THEN              { Convert string to Numeric values }
        CASE GetReal OF
          True  : Val(Worker,RValue,Error);
          False : Val(Worker,IValue,Error)
        END { CASE }
      ELSE
        BEGIN
          RValue := 0.0;
          IValue := 0
        END
    END

END;  { GETString }
```

This routine makes use of the **GotoXY** procedure to locate the cursor on your CRT screen. Turbo Pascal's CRT control procedures are described in Section 18.1. Remember that to use **GotoXY** you must **USE** the **Crt** unit.

GetString begins by drawing a field on the screen. The field consists of two vertical bar characters (ASCII character 124) with periods between them. The number of periods is the maximum length of the string you wish to enter, passed to **GetString** in **MaxLen**:

```
|.........................|
```

This example would be drawn for a **MaxLen** value of 25. The left vertical bar character is located at X,Y on the screen.

GetString can accept a string value to edit in the parameter **XString**. If **XString** has anything in it, those characters are displayed left-justified in the field:

```
|I am a man of letters....|
```

GetString then positions an underscore character for a cursor immediately after the displayed characters, or at the left margin if no characters were displayed:

```
|I am a man of letters_...|
```

At this point **GetString** begins to accept typed characters. No **Read** or **Readln** statements are used in this procedure at all. A DOS call is performed in a tight loop to test for a keypress. If a key was pressed, the **MSDOS** function (see Section 20.5) returns the character pressed; if no key was pressed, it returns character 0.

Once a keypress is accepted, **GetString** decides if it is printable or not. Control characters (ASCII 1-31) are never printable. If the **Numerics** parameter is **True**, only digits, decimal points, the letter E, and minus signs will be accepted as printable. Then, after the string has been completed by the entry of CR, **Val** evaluates the string and places the value in **Value**.

If the **CapsLock** parameter is **True**, lower-case letters will be forced to upper case as they are entered.

Only a few control characters are obeyed. The entry of CR will end string entry and replace the previous contents of **XString** with the entered string. ESC will end string entry but leave **XString** the same as it was on entry. CTRL-X (CANcel) clears the entire string to zero length and erases it from the displayed field.
Entry of BS and DEL destructively backspaces over one character.

Any character that is not printable and not a recognized control character causes the **UhUh** procedure to be invoked (see Section 16.13), signalling to the user that a keypress was ignored. Any attempt to backspace past the left margin of the field will trigger an error signal, as will trying to enter more characters than the field will hold.

Using GetString for Screen Data Entry

GetString is the most complex piece of Pascal code we've examined so far. If you intend to write a lot of programs that interact extensively with the user, you might also find it one of the most useful tools in your software toolbox. It makes the programming of interactive data-entry screens neat and easy.

For many years all computer interaction was done on the "glass teletype" model: Computer and user took turns typing their halves of a dialog on the bottom line of a terminal, with the screen scrolling up one line after each took a turn.

A much tidier option is the notion of a data entry screen. The computer "paints" one or more fields on a cleared screen, and then the user fills in the fields in some well-defined order. The computer uses some reserved portion of the screen (usually the top or bottom lines) to send messages to the user.

The following program is a simple example of a data entry screen, using the **GetString** procedure to provide several fields for the user to fill in. Nothing is done with the information after it is accepted, but in a functional program of this type the data is typically stored in a file.

GetString is another good example of hiding detail in a Pascal program. If you're programming a data entry screen, what's important is what data fields are being entered and what is done with them afterward. The excruciating details of accepting a string character-by-character and, if necessary, converting it into a numeric value are not necessary to understanding the logic of the data entry screen. In this fashion, the details are shunted into a black box named **GetString** where they will not interfere with the clarity of the code that handles the data itself.

```
{--------------------------------------------------------------}
{                           Screen                             }
{                                                              }
{                  Full-screen input demo program              }
{                                                              }
{                              by Jeff Duntemann               }
{                              Turbo Pascal V5.0               }
{                              Last update 7/14/88             }
{                                                              }
{                                                              }
{                                                              }
{--------------------------------------------------------------}


PROGRAM Screen;

USES DOS,Crt,BoxStuff;

{$V-}  { Allow length mismatch for string VAR parameters.  See 23.4 }

CONST
  Capslock      = True;
  NoCapslock   = False;
  Numeric       = True;
  NonNumeric    = False;

TYPE
  String80 = String[80];
  String30 = String[30];
  String6  = String[6];
  String4  = String[4];
  String3  = String[3];

  NAPRec   = RECORD
               Name    : String30;
               Address : String30;
               City    : String30;
               State   : String3;
               Zip     : String6
             END;


VAR
  CH             : Char;
  CurrentRecord : NAPRec;
  Edit           : Boolean;
  Quit           : Boolean;
  Escape         : Boolean;
  WIDTH,HEIGHT   : Integer;
  I,J            : Integer;
  R              : Real;


{$I UHUH.SRC }       { Described in Section 16.11 }
{$I CURSOFF.SRC }    { Described in Section 17.2 }
{$I CURSON.SRC }     { Described in Section 17.2 }
{$I YES.SRC }        { Described in Section 17.2 }
{$I GETSTRIN.SRC }   { Described in Section 15.2 }


PROCEDURE GetScreen(VAR ScreenData : NAPRec;
                        Edit       : Boolean;
                    VAR Escape     : Boolean);

BEGIN
  MakeBox(1,1,79,20,GrafChars);        { Draw the screen box }
  IF NOT Edit THEN WITH ScreenData DO  { If not editing, clear record }
    BEGIN
      Name := ''; Address := ''; City := ''; State := ''; Zip := ''
    END;
  GotoXY(23,2);
  Writeln('<< Name / Address Entry Screen >>');
  WITH ScreenData DO
    BEGIN                        { First draw field frames: }
      GotoXY(5,7);
      Write('>>Customer Name:    |..............................|');
      GotoXY(5,9);
      Write('>>Customer Address: |..............................|');
      GotoXY(5,11);
      Write('>>Customer City:    |..............................|');
      GotoXY(5,13);
      Write('>>Customer State:   |...|');
      GotoXY(5,15);
      Write('>>Customer Zip:     |......| ');
      IF Edit THEN WITH ScreenData DO  { If editing, show current values }
        BEGIN
          GotoXY(26,7);  Write(Name);
          GotoXY(26,9);  Write(Address);
          GotoXY(26,11); Write(City);
          GotoXY(26,13); Write(State);
          GotoXY(26,15); Write(Zip)
        END;                            { Now input/Edit field data: }
      GetString(25,7,Name,30,NoCapslock,NonNumeric,False,R,I,J,Escape);
      IF NOT Escape THEN
        GetString(25,9,Address,30,NoCapslock,NonNumeric,False,R,I,J,Escape);
      IF NOT Escape THEN
        GetString(25,11,City,30,NoCapslock,NonNumeric,False,R,I,J,Escape);
      IF NOT Escape THEN
        GetString(25,13,State,3,Capslock,NonNumeric,False,R,I,J,Escape);
      IF NOT Escape THEN
        GetString(25,15,Zip,6,Capslock,NonNumeric,False,R,I,J,Escape);
    END
END;


BEGIN      { SCREEN MAIN }
  Edit := False;
  CursorOff;
  REPEAT
    ClrScr;
    GetScreen(CurrentRecord,Edit,Escape);  { Input/Edit a data screen }
    IF Escape THEN Quit := True ELSE       { Quit if ESC pressed }
      BEGIN                                { Otherwise summarize data }
        Quit := False;                     { and ask for approval }
        GotoXY(1,22);
        Write('>>Summary: ');
        WITH CurrentRecord DO
          BEGIN
            Write(Name,'/',Address,'/',Zip);
            GotoXY(1,23); Write('>>OK? (Y/N): ');
            IF YES THEN Edit := False ELSE Edit := True
          END
      END
  UNTIL Quit;
  ClrScr;
  CursorOn
END.
```

15.3: MORE EXAMPLES OF STRING MANIPULATION

Perhaps the first ambitious program most beginning programmers attempt is a name/address/phone number manager. Sooner or later, in designing such a program, the problem comes up: How to sort the list on the name field, when names are stored first name first and sorted last name first?

Storing the first name in a separate field is no answer. Suppose you want to store The First National Bank of East Rochester? What is its first name?

The best solution I have found is to store the name last name first, with an asterisk separating the last and first names. For example, Jeff Duntemann would be stored as Duntemann*Jeff. Clive Staples Lewis would be stored as Lewis*Clive Staples. Names maintained in this order are easily sorted by last name. All we need is a routine to turn the inside-out name rightside-in again.

The following routine does just that, and it uses **Pos**, **Copy**, **Delete**, and **Concat**, all in four lines!

```
{<<<< RvrsName >>>>}
{                                                           }
{                                                           }
{                                                           }

PROCEDURE RvrsName(VAR Name : String);

VAR
  TName : String;

BEGIN
  IF Pos('*',Name) <> 0 THEN
    BEGIN
      TName := Copy(Name,1,(Pos('*',Name)-1));
      Delete(Name,1,Pos('*',Name));
      Name := Concat(Name,' ',TName)
    END
END;
```

The theory is simple: If there is no asterisk in the name, it's something like "Granny Maria's Pizza Palace" and needs no reversal, hence the first test. If an asterisk is found, the last name up to, but not including, the asterisk is copied from **Name** into **TName**, a temporary string. Then the last name is deleted from **Name**, up to *and* including the asterisk. What remains in **Name** is thus the first name. Finally, concatenate **TName** (containing the last name) to **Name** with a space to sparate them. The name is now in its proper, first name first form.

A Case Adjuster Function for Strings

Turbo Pascal provides a built-in character function called **UpCase**, predeclared this way:

```
FUNCTION UpCase(Ch : Char) : Char;
```

UpCase accepts a character **Ch** and returns its uppercase equivalent as the function return value. If **Ch** is already uppercase, or a character with no uppercase equivalent, (numerals, symbols, and so on) the character is returned unchanged.

UpCase is a character function, but it suggests that a string function could be built that accepts an arbitrary string value and returns that value converted to uppercase. And although no downcase function exists in Turbo Pascal, an equivalent is not hard to put together. A two way case adjuster function looks like this:

```
{<<<< ForceCase >>>>}
{                                                         }
{                                                         }
{                                                         }

FUNCTION ForceCase(Up : BOOLEAN; Target : String) : String;

CONST
  Uppercase : SET OF Char = ['A'..'Z'];
  Lowercase : SET OF Char = ['a'..'z'];

VAR
  I : INTEGER;

BEGIN
  IF Up THEN FOR I := 1 TO Length(Target) DO
    IF Target[I] IN Lowercase THEN
      Target[I] := UpCase(Target[I])
    ELSE { NULL }
  ELSE FOR I := 1 TO Length(Target) DO
    IF Target[I] IN Uppercase THEN
      Target[I] := Chr(Ord(Target[I])+32);
  ForceCase := Target
END;
```

If you're new to Pascal in general, you may not notice anything strange about this function, but Pascal old-timers will notice that Turbo Pascal is one of the few, if not the only, Pascal implementation that allows user-defined functions to return string values.

In Turbo Pascal, functions may return string values the same as any other values. However, the string type must either be the default type **String** or have been declared before the declaration of your string function. In other words, if you wish your function to return a string with a physical length of 80 you must have declared a string type with that physical length:

```
TYPE
  String80 = String[80];
```

You *cannot* use the bracketed string length notation on a string function return value. That is, you could not have declared **ForceCase** this way:

```
FUNCTION ForceCase(Up      : Boolean;
                   Target : String80) :  String[80];
                                        {^Invalid! }
```

ForceCase will convert all uppercase characters in a string to lowercase, or all lowercase characters in a string to uppercase, depending on the Boolean value of parameter **Up**. If **Up** is true, lowercase is forced to uppercase. Otherwise, uppercase is forced to lowercase. The string **Target** is scanned from character 1 to its last character, and any necessary conversion of character case is done character by character. The "down-case" function is done by taking advantage of the ordering of the ASCII character set, in that lowercase characters have an ASCII value 32 higher than their uppercase counterparts. Add 32 to the ordinal value of an uppercase character, and you have the ordinal value of its lowercase equivalent.

Also note that although the parameter string **Target** is modified during the scan, the modifications are not made to the actual parameter itself, since **Target** was passed by value, not by reference. **ForceCase** received its own private copy of **Target** which it could safely change without altering the "real" **Target**. Refer to Section 14.2 for more on the passing of parameters by value or by reference.

Accessing Command-Line Strings

Most operating systems allow some sort of program access to the command line tail; that is, the optional text that may be typed after the program name when invoking a program from the operating system command prompt:

```
A>CASE DOWN B:FOOFILE.TXT
```

In this example, the characters typed after the program name "CASE" constitute the command line tail:

```
DOWN B:FOOFILE.TXT
```

Turbo Pascal provides a very convenient (and operating system independent) method of gaining access to the command tail. Two predefined functions are connected with the command line tail: **ParamCount** and **ParamStr**. They are predeclared this way:

```
FUNCTION ParamCount : Integer;
FUNCTION ParamStr(ParameterNumber : Integer) : String;
```

The function **ParamCount** returns the number of parameters typed after the command on the operating system command line. Parameters must have been separated by spaces or tab characters to be considered separate parameters. Commas, slashes, and other symbols on will *not* delimit separate parameters!

ParamStr returns a string value that is one of the parameters. The number of the parameter is specified by **ParameterNumber**, starting from 1. If you typed several parameters on the command line, for example,

```
ParamStr(2)
```

will return the second parameter.

Keep this in mind: you *must* read the command line tail before opening your first disk file! The same area use to store the tail is also used in buffer disk accesses in some cases. The best way to do this is to keep an array of strings large enough to hold the maximum number of parameters your program needs, and read the parameters into the array as soon as your program begins running. This is easy enough to do:

```
VAR
  I : Integer;
  ParmArray : ARRAY[1..8] OF String[80];


FOR I := 1 TO ParamCount DO
  ParmArray[I] := ParamStr(I);
```

Now you have the parameters safely in **ParmArray** and can examine and use them at your leisure.

For examples of **ParamCount** and **ParamStr** in use, refer to the **HexDump** program in Section 19.11 and the **Caser** program in Section 19.7.

16 Standard Functions

The ISO Standard Pascal definition includes a number of standard functions that are built into the language and need not be declared and coded into your program. These functions fall into two basic groups: Mathematical functions, which provide fundamental operations such as square and square root, absolute value, natural logarithms, and trig functions; and transfer functions, which define relationships between otherwise incompatible data types like **Integer** and **Char**.

Many books refer to the parameter passed to a standard function as its *argument.* This borrows jargon from the world of mathematics and may be confusing, because some people will wonder what the difference is between an argument and a parameter. There is no difference other than the term. To lessen the confusion, I will use the term parameter, which we have been using with respect to functions all along.

All of the standard functions described in this section may accept expressions as parameters, as long as those expressions evaluate to a value of the correct type. In other words, you may say **Sqrt(Sqr(X)+Sqr(Y))** as well as **Sqrt(16)**. Just make sure you don't try to extract the square root of a **Boolean** value, or of an enumerated type, and so on.

Turbo Pascal implements all the standard functions from ISO Pascal, and quite a few of its own. In this section we'll discuss them in detail.

16.1: ROUND AND TRUNC

Round and **Trunc** are fence sitters. They are both mathematical functions, in the sense that they provide a mathematical service, and they are also transfer functions, in that they provide a bridge between the partly incompatible types **Real** and **Integer**.

We have already seen in Section 12.2 that any integer value may be assigned to a variable of type **Real**. But the reverse is not true, since a **Real** value may have a decimal part, and there is no way to express a decimal part in type **Integer**. **Round** and **Trunc** give us our choice of two ways to "transfer" a **Real** value into an **Integer** value. **Round** and **Trunc** both accept parameters of type **Real** and return values that may be assigned to either type **Integer** or **Real**.

Round

In mathematics, *rounding* a number means moving its value to the nearest integer. This is the job done by **Round**. **Round(X)** returns an integer value that is the integer closest to **X**. The direction in which a real number with a fractional part is rounded is usually given as *up* or *down.* This can be confusing when you start dealing with negative real numbers. I prefer to visualize a number line and speak of *toward* 0 or *away from* 0.

For **X** *greater than* 0: Rounds *away from 0* (up) for fractional parts greater than or equal to .5. Rounds *toward 0* (down) for fractional parts less than .5.

For **X** *less than* 0: Rounds *away from 0* (down) for fractional parts greater than or equal to .5. Rounds *toward 0* (up) for fractional parts less than .5.

Some examples:

```
Round(4.449)          { Returns 4  }
Round(-6.12)          { Returns -6 }
Round(0.6)            { Returns 1  }
Round(-3.5)           { Returns -4 }
Round(17.5)           { Returns 18 }
```

Because the way rounding works with **Round** is symmetric with respect to 0, **Round(-X)** is equal to **-Round(X)**.

Note that using **Round(X)** for **X > MaxInt** will return a value that cannot be assigned to an integer without generating a runtime error.

```
Error 201: Range check error
```

This will occur if range checking is enabled; if range checking is *not* enabled, your program will continue executing, but the value actually assigned to the integer will be unpredictable.

Trunc

Truncating a real number simply means removing its fractional part and dealing with what's left. **Trunc(X)** returns the closest integer value *toward* 0. If you ponder that for a moment you'll see that it is equivalent to removing the fractional part and calling the whole number part an integer. Examples:

```
Trunc(17.667)         { Returns 17 }
Trunc(-3.14)          { Returns -3 }
Trunc(6.5)            { Returns 6  }
Trunc(-229.00884)     { Returns -229 }
```

Turbo Pascal 4.0 and 5.0 remove a restriction imposed by V3.0 that prevented the programmer from using **Round** and **Trunc** to round or truncate large real numbers, even if you intended to assign the resulting value to type **Real**. Turbo Pascal 3.0 actually worked with intermediate integer values internally, and would trigger a range error if **Round** or **Trunc** were passed a parameter greater than **MaxInt**. This is no longer true. Assuming the type to which you assign the value returned by **Round** or **Trunc** has the range to contain the value, there will be no problems. When returning values to real numbers, **Trunc** is now the equivalent of the Turbo Pascal function **Int.**

16.2: SQR AND SQRT

There is nothing complicated here. **Sqr(X)** squares **X**. It is completely equivalent to **X * X**, and Pascal includes it not only because squaring is done so frequently in mathematics, but also because (as we will discuss later) there is no exponentiation operator in Pascal and hence no clean notation for **X** raised to a power of two.

Sqr may operate on both integers and reals. If you square an integer with **Sqr**, the returned value is an integer. If you square a real with **Sqr**, the returned value is real.

Sqrt(X) may also operate on either an integer or real **X**, but the value returned is *always* type **Real**.

A few examples:

```
CONST
  PI = 3.14159;

VAR
  I : Integer;
  R : Real;

I := 64;
R := 6.077;

Sqrt(16)        { Returns 4.0; a real number! }
Sqrt(PI)        { Returns 1.77245 }
Sqrt(I)         { Returns 64; real number }
Sqr(2.4)        { Returns 4.8; again, real }
Sqr(7)          { Returns 49; integer or real }
Sqr(I)          { Returns 4096; integer }
Sqr(R)          { Returns 36.92993; real }
```

The following procedure calculates the length of the hypotenuse of a right triangle, given the other two sides:

```
FUNCTION Hypotenuse(Side1,Side2 : Real ) : Real;

BEGIN
  Hypotenuse := Sqrt(Sqr(Side1) + Sqr(Side2))
END;
```

The algorithm, of course, is the Pythagorean Theorem.

16.3: TRIGONOMETRIC FUNCTIONS

There are three trigonometric functions among the standard functions of Pascal: **Sin**, **Cos**, and **ArcTan**. Where are **Tan**, **ArcSin**, and all the others? Well, given **Sin**, **Cos**, and **ArcTan**, all other trigonmetric functions are easily derived. Insisting that they be built into the compiler would make the compiler more complex and prone to errors. It would also make the Pascal language more cluttered than it has to be.

Sin(X), **Cos(X)**, and **ArcTan(X)** all return real results. **X** may, however, be an integer or a real number. You should note well that *X represents radians, not degrees.* A radian equals 57.29578 degrees. Radians, however, are usually thought of in terms of pi (3.14159) and fractions of pi. Thus, 360 degrees = 2pi radians; 180 degrees = pi radians, and so forth. Pascal's trigonometric functions behave as you would expect them to behave from textbook discussions of trigonometry.

Deriving Other Trig Functions

With **Sin**, **Cos**, and **Arctan**, one can build functions returning all other trigonometric relationships. For example:

```
FUNCTION Tan(X : Real) : Real;

BEGIN
  Tan := Sin(X) / Cos(X)
END;


FUNCTION Cot(X : Real) : Real;

BEGIN
  Cot := Cos(X) / Sin(X)
END;

FUNCTION Sec(X : Real) : Real;

BEGIN
  Sec := 1 / Cos(X)
END;


FUNCTION Csc(X : Real) : Real;

BEGIN
  Csc := 1 / Sin(X)
END;
```

Note that even though the **X** parameter passed to these functions is declared as type **Real**, Turbo Pascal will allow you to pass an integer literal or variable in **X** without error, and will treat the value as a real number without a fractional part during the calculations.

16.4: ABSOLUTE VALUE, NATURAL LOGS, EXPONENTS

Absolute Value

Absolute value in mathematics is the distance of a number from 0. In practical terms, this means stripping the negative sign from a negative number and leaving a positive number alone. The Pascal function **Abs(X)** returns the absolute value of **X**. **X** may be type **Real** or **Integer**. The type of the returned value is the same as the type of **X**. For example:

```
Abs(-61)         { Returns 61; type Integer }
Abs(484)         { Returns 484; also Integer }
Abs(3.87)        { Returns 3.87; type Real }
Abs(-61.558)     { Returns 61.558; also Real }
```

The **Abs** function is actually a shorthand form of the following statement:

```
IF X < 0 THEN X := - X;
```

Natural Logarithms

There are two Pascal standard functions that deal with *natural logarithms*. Natural logarithms are mathematical functions that turn on a remarkable irrational number named ***e***, which, to six decimal places, is 2.718282. Explaining where ***e*** comes from, or explaining natural logarithms in detail, is somewhat outside the charter of this book. Do read up on them (in any senior high math text) if the concept is strange to you.

In any case, **Exp(X)** returns the exponential function for **X**. **X** may be a real number or an integer, but the returned value is always real. The exponential function raises ***e*** to the **X** power. Therefore, when you evaluate **Exp(X)**, what you are actually evaluating is e^X.

Ln(X) returns the natural logarithm (logarithm to the base ***e***) of **X**. Again, **X** may be type **Integer** or **Real**, and the returned value, as well, is always type **Real**. The sense of the **Ln(X)** function is the reverse of **Exp(X)**: Evaluating **Ln(X)** yields the exponent to which ***e*** must be raised to give **X**.

Natural logarithms are the most arcane of Pascal's mathematical standard functions. They are most used in mathematics that many of us would consider "heavy." However, there is one use for which natural logarithms fill an enormous hole in Pascal's definition: Exponentiation. Unlike most languages, Pascal contains no general function for raising X to the Yth power. In FORTRAN, for example, the exponentiation operator is the double asterisk: **X**Y** raises **X** to the **Yth** power. **Exp** and **Ln** allow us to create a function that raises one number to a given power:

```
FUNCTION Power(Mantissa, Exponent : Real) : Real;

BEGIN
  Power := Exp(Ln(Mantissa)*Exponent)
END;
```

This almost certainly looks like magic unless you really understand how natural logarithms work. Two cautions: The result returned is type **Real**, not **Integer**. Also, do not pass a 0 or negative value to **Mantissa**. Runtime error 4: Ln argument error will result. Reason: **Ln(X)** for a negative **X** is undefined!

16.5: ORD AND CHR

The functions **Ord** and **Chr** are true transfer functions, providing you with a well-documented, "legal" pathway between the otherwise incompatible types **Integer** and **Char**. **Ord** actually provides the pathway between integers and *any* ordinal type, hence the name.

Ord

As its name suggests, **Ord(X)** deals with ordinal types. Ordinal types are those types that can be *enumerated;* that is, types with a fixed number of values in a well-defined order.

Ord(X) returns the ordinal position (an integer) of the value X in its ordinal type. The sixty-sixth character in the ASCII character set is the capital letter **'A'**. **Ord('A')** returns 65. The third color in our old friend type **Spectrum** is **Yellow**. **Ord(Yellow)** returns 2. Remember that, for both examples, we start counting at 0!

Chr

Chr goes in the opposite direction from **Ord**: **Chr(X)** returns a character value corresponding to the **X**th character in the ASCII character set (**X** is an integer). **Chr(65)** returns the capital **'A'**. **Chr(66)** returns capital letter **'B'**, and so on. The most important use of **Chr(X)** is generating character values that are not expressed by any symbol that you can place between single quote marks. How do you put a line feed in quotes; or worse yet, a bell character? You don't, you express them with **Chr**:

```
Chr(13)        { Returns ASCII carriage return (CR) }
Chr(7)         { Returns ASCII bell (BEL)   }
Chr(127)       { Returns ASCII delete (DEL) }
Chr(8)         { Returns ASCII backspace (BS) }
```

Chr allows you to return a character based on an integer expression. The procedure **CapsLock**, for example, uses **Ord** and **Chr** to translate a character into an integer, manipulate the integer, and then translate the integer back into a character:

```
PROCEDURE CapsLock(VAR Target : String);

VAR Lowercase : SET OF Char;
    I         : Integer;

BEGIN
  Lowercase := ['a'..'z'];
  FOR I := 1 TO Length(Target) DO
    IF Target[I] IN Lowercase THEN
      Target[I] := Chr(Ord(Target[I]) - 32)
END;
```

Parenthetically, Turbo Pascal has a built-in procedure, **UpCase**, which will accomplish the same thing as the expression:

```
Chr(Ord(Target[I]) - 32)
```

but the procedure **CapsLock** as given above will compile under ISO Standard Pascal.

Don't try to pass **Chr** an integer value higher than 255. The results will be undefined, probably garbage.

16.6: PRED AND SUCC

We discussed these two standard functions, in connection with **FOR** loops, informally in Section 13.4. Now it's time for a closer look.

One of the properties of an ordinal type is that its value exists in a fixed and well-defined order. In other words, for type **Integer**, 3 comes after 2, not before. For type **Char**, 'Q' follows 'P' which follows 'O', and so on. The order is always the same.

This order is called the "collating sequence" or "collating order" of an ordinal type. Given a value of an ordinal type, **Ord** tells you which position that value occupies in its collating sequence. Given a value of an ordinal type, **Pred** and **Succ** return the next value before that value or after that value, respectively, such that:

```
Pred('Z')       { Returns 'Y' }
Succ('w')       { Returns 'x' }
Pred(43)        { Returns 42 }
Succ(19210)     { Returns 19211 }
Pred(Orange)    { Returns Red }
Succ(Green)     { Returns Blue }
Pred(Red)       { Undefined! }
```

The last example bears a closer look. The predecessor value of **Red** is undefined. You will recall the definition of our enumerated type **Spectrum**:

```
Spectrum = (Red,Orange,Yellow,Green,Blue,Indigo,Violet);
```

Red is the very first value in the type. There is nothing before it, so **Pred(Red)** makes no sense in the context of type **Spectrum**. Similarly, **Succ(Violet)** makes no sense, since there is no value in **Spectrum** after **Violet**.

Pred(<value>) of the first value of an ordinal type is undefined. **Succ(<value>)** of the last value of an ordinal type is undefined.

Pred and **Succ** provide a means of "stepping through" an ordinal type to do some repetitive manipulation on a range of the values in that ordinal type. For example, in printing out the names on a telephone/address list, we might want to put a little header before the list of names beginning with A, and then before the list of names beginning with B, and so on. Assuming that the names are stored in sorted order in an array, we might work it this way:

```
VAR
  Names       : ARRAY[1..200] OF String[35];
  NameCount : Integer;


PROCEDURE Header(Ch : Char);

BEGIN
  Write(LST,Chr(13),Chr(10));
  Writeln(LST,'[',Ch,']--------------------------------')
END;


PROCEDURE PrintBook(NameCount : Integer);

VAR I     : Integer;
    Ch    : Char;
    AName : String[35];

BEGIN
  Ch := 'A';
  Header(Ch);
  FOR I := 1 TO NameCount DO
    BEGIN
      AName := Names[I];
      IF AName[1] <> CH THEN
        REPEAT
          CH := Succ(CH);
          Header(Ch)
        UNTIL AName[1] = CH;
        Writeln(LST,AName)
      END
  END;
```

Assume that the array **Names** has been filled somehow with names, and that the number of names has been placed in **NameCount**. The names must be in sorted order, last name first. When **PrintBook** is invoked, the list of names in **Names** is printed on the system printer with a header for each letter of the alphabet:

```
[A]--------------------------------
Albert*Eddie
Aldiss*Brian
Anselm*Jo
Anthony*Piers

[B]--------------------------------
Brooks*Bobbie
Bentley*Mike

[C]--------------------------------
Chan*Charlie
Charles*Ray
Cabell*James Branch

[D]--------------------------------

[E]--------------------------------

[F]--------------------------------
Farmer*Philip Jose
Foglio*Phil
Flor*Donna
```

and so on. Letters for which no names exist in the list will still have a printed header on the list. The printed listing, if cut into memo-sized sheets, would make the core of a "little black book" for names and addresses.

Look at the listing of **PrintBook**. **Ch** is given an intial value of A. As the names are printed, the first letter of each name is compared to the letter stored in **Ch**. If they don't match, the loop:

```
REPEAT
  Ch := Succ(Ch);
  Header(Ch)
UNTIL AName[1] = Ch;
```

is executed. The letter in **Ch** is "stepped" along the alphabet until it "catches up" to the first letter in **AName**. For each step along the alphabet, a header is printed.

We might have written **Ch := Chr(Ord(Ch)+1)** instead of **Ch := Succ(Ch)**. **Succ** provides a much crisper notation. Furthermore because **Chr** does not work with enumerated types, **Succ** is the only standard way to step along the values of a programmer-defined enumerated type like **Spectrum.**

16.7: ODD

The last of the standard functions from ISO Pascal is a transfer function: **Odd(X)**. **X** is an integer value. If **X** is an odd value, **Odd(X)** returns the Boolean value **True**. If **X** is an even value, **Odd(X)** returns **False**.

Odd is thus a way of expressing an integer value as a Boolean. Any even number can express the value **False**, and any odd number can express the value **True**. In this case, 0 is considered an even number by virtue of its position between two odd numbers, 1 and −1.

Up to this point, all the functions described have been present in ISO Standard Pascal. Turbo Pascal provides a number of other built-in functions that ISO Standard Pascal does not.

16.8: FRAC AND INT

With these two functions you can separate a real number into its whole number part and its fractional part.

Frac(R) returns the fractional part of real number **R**. In other words, **Frac(24.44789)** would return 0.44789.

Int(R) returns the whole number part of real number R. In other words, **Int(241.003)** would return 241.0.

Both **Frac** and **Int** return values of type **Real**. You cannot directly assign the return values from either of these functions to an integer type. For returning the whole number part of a real number to an integer type, use **Trunc** instead.

16.9: HI, LO, AND SWAP

Type **Integer**, if you recall, is represented in memory as two bytes. There are times when it is necessary to obtain the value of each of the two bytes apart from its brother. Functions **Hi** and **Lo** allow you to do this:

Hi(I) returns the value of the high order byte of **I**. Technically, what happens is that the value of **I** is shifted eight bits to the right, so that the high order byte becomes the low order byte. The high order byte is filled with 0 bits. For example:

Hi(256) returns 1. Why? Can you picture it happening?
Hi(17341) returns 67.
Hi(-10366) returns 215. How does the negative sign figure in?
Lo(I) returns the low order byte of **I**. All that happens here is that the high eight bits of **I** are forced to zero. For example:
Lo(256) returns 0.
Lo(17341) returns 189.
Lo(-10366) returns 130. Can **Lo** ever return a negative number?

Swap(I) *exchanges* the two bytes of integer **I**. The high order byte becomes the low order byte, and the low order byte becomes the high order byte. For example:
Swap(17341) returns -17085. Where does the negative sign come from?
Swap($66FF) returns $FF66. Hexadecimal notation makes things a good deal clearer!

Hi, **Lo**, and **Swap** find good use when you must move values into and out of CPU registers. Many 8086 registers are 2-bytes wide and must be filled all at once; yet many operating system calls require that the two halves of a CPU register be filled with unrelated bytes of data. Furthermore, after a DOS call the two bytes of a register may contain necessary but unrelated data, and must be separated.

Suppose, for example, that you need to pass $01 to register AH and $06 to register AL. AH and AL are two halves of 8086 register AX, and AX must be passed to the operating system as a unit. Turbo Pascal treats AX as an integer, part of an important data structure called **Registers** (see Section 20.3). How does one load two separate numbers into one integer? Try this:

```
VAR
  AX,JMask,QBit : Integer;

JMask := $06;          { The high bytes of both these }
QBit  := $01;          { variables are zero! }
AX    := QBit;         { QBit goes into the low byte of AX; }
AX    := Swap(AX);     { then Swap flips low byte for high; }
AX    := AX + JMask;   { so that adding JMask to AX puts it }
                       { into AX's low byte. }
```

Hi and **Lo** comes into play when the operating system returns a value to your program in register **AX**. You want to extract **JMask** and **QBit** from **AX** again this way:

```
JMask := Lo(AX);       { JMask comes back in AX's low byte; }
QBit  := Hi(AX);       { and QBit in AX's high byte }
```

The need for **Hi**, **Lo**, and **Swap** will become more apparent after you understand what is involved in making DOS calls and software interrupts (see Section 20.2).

16.10: PI

Turbo Pascal also includes a standard function **Pi** that returns the value of pi to as many significant figures as the variable to which it returns the value may express. The value returned is a real number type and so may not be assigned to any integer type, but, as with any real number type, it may be assigned to the coprocessor-based types **Single**, **Double**, **Extended** and **Comp**.

You should keep in mind that differences in precision in the type receiving a value from **Pi** will affect the exact decimal values returned, although for all but the most exacting requirements these differences can be ignored.

To illustrate, the following code displays the actual values returned by function **Pi** to the various real number types:

```
VAR
  RS : Single;
  RR : Real;
  RD : Double;
  RE : Extended;
  RC : Comp;


  RS := Pi; RR := Pi; RD := Pi; RE := Pi; RC := Pi;
  Writeln('Type Single:   ',RS:4:25);
  Writeln('Type Real:     ',RR:4:25);
  Writeln('Type Double:   ',RD:4:25);
  Writeln('Type Extended: ',RE:4:25);
  Writeln('Type Comp:     ',RC:4:25);
```

And here is the screen output you'll see:

```
Type Single:   3.141592741012573240
Type Real:     3.1415926535883045330
Type Double:   3.1415926535897931200
Type Extended: 3.1415926535897932400
Type Comp:     3.0000000000000000000
```

Note that type **Comp** can accept a value from **Pi** but will not store the fractional part.

16.11: INC AND DEC

Turbo Pascal 4.0 adds two new standard procedures to its already considerable toolkit: **Inc** and **Dec**. Both operate only on ordinal types and their function is simple: **Inc** increments an ordinal value, and **Dec** decrements an ordinal value. Functionally, they are equivalent to **Pred** and **Succ** *except* that **Pred** and **Succ** are functions rather than procedures.

In other words, **Inc** and **Dec** may stand alone as statements:

```
VAR
  I : Integer;
  Ch : Char;

I := 17;
Ch := 'A';
Inc(I);       { I now contains 16 }
Dec(Ch);      { Ch now contains '@' }
```

By contrast, **Pred** and **Succ** need to be placed on the right side of an assignment statement, either alone or inside of an expression:

```
I := 17;
Ch := 'A';
I := Succ(I);     { I now contains 17 }
Ch := Pred(Ch);   { Ch now contains '@' }
```

Why use **Inc** and **Dec** if the standard and much more portable **Pred** and **Succ** are available? Only one reason: speed. To understand this, you need to consider what the compiler has to do when it generates native 8086 code from a Turbo Pascal statement. An assignment statement like:

```
I := Succ(I);
```

is treated as any assignment statement: First the compiler generates code to evaluate the expression on the right side of the assignment statement, and then it moves that intermediate value to replace the current value of the variable on the left side of the assignment statement. Moving things from one location in memory to another takes a fair amount of time compared to the time it takes simply to increment or decrement a variable that fits neatly in either one or two 8086 registers.

A statement like:

```
Inc(I);
```

on the other hand, generates code that goes directly to the location of variable **I** and increments it. Nothing unnecessary is done, and the code generated is faster than the code generated for **Succ**. It may not seem a *lot* faster, and it won't be noticeable until you need to increment something thousands of times in succession within a tight loop. Then, all those microscopic fragments of wasted time add up to significant program delays.

Like so many other things in life, it's a tradeoff. You, the program developer, will have to decide whether the gain in speed is worth the loss in portability, or vice versa.

16.12: RANDOM NUMBER FUNCTIONS

Built into Turbo Pascal are two functions that return pseudorandom numbers, **Random** and **Random(I)**. **Random** returns a real number, and **Random(I)** returns an integer.

Random returns a pseudorandom number of type **Real** that is greater than or equal to zero and less than one. This statement:

```
FOR I := 1 TO 5 DO Writeln(Random);
```

might display:

```
7.0172090270E-01
7.3332131305E-01
8.0977424840E-01
6.7220290820E-01
9.2550002318E-01
```

The E-01 exponent makes all these numbers fall in the range 0.0 through 0.9999999999. All random numbers returned by the function **Random** fall within this range, but of course if you need random real numbers in another range, you need only shift the decimal point the required number of places to the right.

Random(I) returns random integers. The parameter is an integer that sets an upper bound for the random numbers returned by the function. **Random(I)** will return a number greater than or equal to 0 and less than **I**. **I** may be any integer up to **MaxInt**. **I** may be negative. However, making **I** *any* negative value has the same effect as passing **MaxInt** to **Random(I)**. There is no way to make **Random(I)** return negative random numbers.

There is frequently a need for a random number in a particular range, say between 15 and 50 or between 100 and 500. The procedure **Pull** meets this need by extracting random integers until one falls in the range specified by **Low** and **High**.

```
{<<<< Pull >>>>}
{                                                   }
{                                                   }
{                                                   }


FUNCTION Pull(Low,High : Integer) : Integer;

VAR
  I : Integer;
```

```

BEGIN
  REPEAT                    { Keep requesting random integers until }
    I := Random(High + 1); { one falls between Low and High }
  UNTIL I >= Low;
  Pull := I
END;
```

Randomize

The **Randomize** procedure exists because Turbo Pascal's random numbers, like all random numbers generated in software, are not random at all but only *pseudorandom,* which means that a series of such numbers approximates randomness. The *series* of such numbers may well repeat itself each time the program is run, unless the random number generator is "reseeded" with a new seed value. This is the job of **Randomize**.

Randomize should be called at least once in every program, and to make your pseudorandom numbers more nearly random, might be called each time you want a new random number or series of random numbers.

Randomize did not work correctly in Turbo Pascal releases 1 and 2. In release 3.0 and later, **Randomize** appears to work correctly.

A Dice Game

The following program shows one use of random numbers in a game situation. **Rollem** simulates the roll of one or more dice, up to as many as will fit across the screen. Procedure **Roll** may be placed in your function/procedure library and used in any game program that must roll dice in a visual manner. It will display a number of dice at location **X,Y** on the screen, where **X,Y** are the coordinates of the upper left corner of the first die. The **NumberOfDice** parameter tells **Roll** how many dice to roll; there is built-in protection against attempting to display more dice than space to the right of **X** will allow.

Rollem is also a good exercise in **REPEAT/UNTIL** loops. The **MakeBox** procedure was described in Section 14.1. In this version of **Rollem**, the values for the fields in the box-draw record **GrafChars** are filled via assignment statements. They could as well be read from a file. How would you go about making **GrafChars** a record constant, initialized with values correct for your computer?

```
{--------------------------------------------------------------}
{                          Rollem                              }
{                                                              }
{  A dice game to demonstrate random numbers and box draws     }
{                                                              }
{                           by Jeff Duntemann                  }
{                           Turbo Pascal V5.0                  }
{                           Last update 7/14/88                }
{                                                              }
{                                                             }
{                                                             }
{-------------------------------------------------------------}

PROGRAM Rollem;

USES Crt,BoxStuff;

CONST
  DiceFaces : ARRAY[0..5,0..2] OF STRING[5] =
                  (('     ','  o  ','     '),  { 1 }
                   ('o    ','     ','    o'),  { 2 }
                   ('    o','  o  ','o    '),  { 3 }
                   ('o   o','     ','o   o'),  { 4 }
                   ('o   o','  o  ','o   o'),  { 5 }
                   ('o o o','     ','o o o')); { 6 }


TYPE
  String80  = String[80];


VAR
  I,X,Y          : Integer;
  Width,Height   : Integer;
  Quit           : Boolean;
  Dice,Toss      : Integer;
  DiceX          : Integer;
  Ch             : Char;
  Banner         : String80;



PROCEDURE Roll(X,Y           : Integer;
               NumberOfDice : Integer;
               VAR Toss     : Integer);

VAR I,J,Throw,XOffset : Integer;

BEGIN
  IF (NumberOfDice * 9)+X >= 80 THEN      { Too many dice horizontally    }
    NumberOfDice := (80-X) DIV 9;         { will scramble the CRT display! }
  FOR I := 1 TO NumberOfDice DO
    BEGIN
      XOffset := (I-1)*9;                 { Nine space offset for each die }
      MakeBox(X+XOffset,Y,7,5,GrafChars); { Draw a die }
      Throw := Random(6);                 { "Toss" it  }
      FOR J := 0 TO 2 DO                  { and fill it with dots }
        BEGIN
          GotoXY(X+1+XOffset,Y+1+J);
          Write(DiceFaces[Throw,J])
        END
    END
END;



BEGIN
```

```
  Randomize;                     { Seed the pseudorandom number generator }
  ClrScr;                        { Clear the entire screen }
  Quit := False;                 { Initialize the quit flag }
  Banner := 'GONNA Roll THE BONES!';
  MakeBox(-1,1,Length(Banner)+4,3,GrafChars);          { Draw Banner box }
  GotoXY((80-Length(Banner)) DIV 2,2); Write(Banner);  { Put Banner in it }
  REPEAT
    REPEAT
      FOR I := 6 TO 18 DO        { Clear the game portion of screen }
        BEGIN
          GotoXY(1,I);
          ClrEol
        END;
      GotoXY(1,6);
      Write('>>How many dice will we Roll this game? (1-5, or 0 to exit): ');
      Readln(Dice);
      IF Dice = 0 THEN Quit := True ELSE  { Zero dice sets Quit flag }
        IF (Dice < 1) OR (Dice > 5) THEN  { Show error for dice out of range }
          BEGIN
            GotoXY(0,23);
            Write('>>The legal range is 1-5 Dice!')
          END
    UNTIL (Dice >= 0) AND (Dice <= 5);
    GotoXY(0,23); ClrEol;        { Get rid of any leftover error messages }
    IF NOT Quit THEN             { Play the game! }
      BEGIN
        DiceX := (80-(9*Dice)) DIV 2;   { Calculate centered X for dice }
        REPEAT
          GotoXY(1,16); ClrEol;
          Roll(DiceX,9,Dice,Toss);                        { Roll & draw dice }
          GotoXY(1,16); Write('>>Roll again? (Y/N): ');
          Readln(Ch);
        UNTIL NOT (Ch IN ['Y','y']);
        GotoXY(1,18); Write('>>Play another game? (Y/N): ');
        Readln(Ch);
        IF NOT (Ch IN ['Y','y']) THEN Quit := True
      END
  UNTIL Quit      { Quit flag set ends the game }
END.
```

16.13: SOUND ROUTINES

Turbo Pascal contains two procedures that control sound production from the PC's speaker. These two procedures, **Sound** and **NoSound**, plus the built-in procedure **Delay** enable you to generate specified frequencies for specified periods of time.

Sound(Frequency) initiates a sound with a frequency given as Hertz (cycles per second) in **Frequency**. **Sound** says nothing about how long the sound will be generated; it will remain on until turned off by **NoSound**.

NoSound turns the speaker off again.

As a simplest case example, to generate a 1 Khz (kilohertz) tone for one second, these three statements would be required:

```
Sound(1000);          { Initiate sound at 1000 Hertz }
Delay(1000);          { Delay for 1000 milliseconds }
NoSound;              { Turn sound off again }
```

Breaking away from the proverbial beep as a signal from your programs is easy enough once you convince yourself that creative noise is not exclusively for video games. One simple signal I use is an audio "uh-uh" to signal that something is not right, especially character input from the keyboard:

```
{<<<< UhUh >>>>}
{                                                   }
{                                                   }
{                                                   }

PROCEDURE UhUh;

VAR
  I : Integer;

BEGIN
  FOR I := 1 TO 2 DO
    BEGIN
      Sound(50);
      Delay(100);
      NoSound;
      Delay(50)
    END
END;
```

Type this one in and try it. You'll get the idea.

Another interesting and very simple application of Turbo Pascal's sound support is shown below. It's a beeper with a difference—it sounds like those fancy telephones you hear in lawyers' offices.

```
{<<<< Beep >>>>}
{                                                   }
{                                                   }
{                                                   }

PROCEDURE Beep;

VAR
  I : Integer;

BEGIN
  FOR I := 1 TO 3 DO
    BEGIN
      Sound(800);
      Delay(50);
      Sound(500);
      Delay(50)
    END;
  NoSound;
END;
```

Earlier versions of Turbo Pascal had the problem that **Sound** and **Delay** were dependent on clock speed. In other words, on faster machines, a given tone generated by **Sound** would be higher than on a slower machine, and the time delay generated with **Delay** would vary depending on a machine's clock speed as well. With Turbo Pascal 3.0 and later, these problems have been corrected, and both routines are now acceptably independent of clock speed.

A Morse Code Generator

Another example of both **Sound** and **Delay** in use lies in the generation of Morse code characters. Procedure **SendMorse** accepts strings containing plain ASCII text, and it will use the computer speaker to transmit the code audibly. Two additional parameters specify the code speed in words per minute (WPM) and the frequency of the speaker tone used, in Hertz.

The amateur radio practice of blending two separate Morse characters together for certain purposes (such as SK, meaning "end of contact") is handled by prefixing the pair to be "blended" with an asterisk, '*', a character for which there is no common Morse equivalent. The blended SK would then be passed to **SendMorse** as **"*SK"**.

Local procedure **Morse** accepts strings containing information already encoded as "dits" and "dahs", where the ASCII period character '.' represents a dit and the ASCII dash '-' represents a dah. **Morse** was set out apart from the body of **SendMorse** so that it could be rewritten to output the code through some other port than the audible speaker. One obvious place is through the PC's cassette relay, as described above.

SendMorse uses the ARRL (American Radio Relay League) published definition for WPM as

```
WPM = 1.2 * B
```

where B is the baud rate of the transmission; essentially the rate at which the smallest units of information are passed. This unit is the length of the audible portion of a "dit," though it must be remembered that the "dit" properly includes the "dit"-length silence following it. The baud rate is thus *twice* the rate of an uninterrupted series of "dits." The length of each "dit" can be derived from the above formula as:

```
                 1.2
DitLength =   ---------
                 WPM
```

which, when multiplied by 1000, becomes an integer that can be passed to the **Delay** procedure as the integer number of milliseconds by which to delay.

The **Delay** procedure is tolerably clock speed independent, and **SendMorse** will transmit code within a few percent of the passed code speed figure regardless of the speed of the host computer. I have tested this on a 4.77-Mhz PC and a 16-Mhz 80386

AT compatible, and the speeds are close enough to count on for code practice leading to the FCC amateur exams.

```
{->>>>SendMorse<<<<----------------------------------------------}
{                                                                }
{ Filename : SENDMORS.SRC -- Last Modified 7/2/88                }
{                                                                }
{ This procedure converts plain text to audible Morse Code at    }
{ a specified code speed and tone frequency.  The times are      }
{ fairly accurate and have been tested on a 4.77 Mhz PC and a    }
{ 16 Mhz 80386 machine without significant difference in         }
{ code speed.  (Turbo's V3.0 DELAY procedure is finally clock    }
{ speed independent.)  The useful range of the procedure is      }
{ from 10-35 WPM, with best "feel" at 15-25 WPM.                 }
{                                                                }
{ Text is passed to SendMorse as quoted strings of plain         }
{ text.  If two characters are to be sent as one without an      }
{ intermediate delay, an asterisk ('*') must precede the two     }
{ characters.  This replaces the overbar used in most amateur    }
{ literature.                                                    }
{                                                                }
{                                                                }
{                                                                }
{----------------------------------------------------------------}

PROCEDURE SendMorse(PlainText : String;
                    ToneFrequency : Integer;
                    CodeSpeed     : Integer);

VAR
  I             : Integer;
  ToneLength    : Integer;
  DitLength     : Integer;
  CodeChar      : String;
  BlendNextTwo  : Boolean;


{ Code is passed to local procedure Morse as literal dots and   }
{ dashes:  '-.-.' = 'C', and so on.  SendMorse converts from    }
{ text to the dot/dash code representation. }

PROCEDURE Morse(CodeChar : String);

VAR
  I : Integer;

BEGIN
  FOR I := 1 TO Length(CodeChar) DO
    BEGIN
      IF CodeChar[1] IN ['.','-'] THEN
        BEGIN
          IF CodeChar[I] = '.' THEN ToneLength := DitLength
            ELSE ToneLength := DitLength * 3;
          Sound(ToneFrequency);
          Delay(ToneLength);
          NoSound;
          Delay(DitLength)
        END
    END
END;

BEGIN
  BlendNextTwo := False;
  { Code speed calculation is derived from formulae published in }
  { the 1986 ARRL Handbook, Section 9-8.   I recommend running   }
  { this procedure at 10 WPM or greater; 15-20 WPM is its most   }
  { effective range.  Timer resolution interferes above 35 WPM.  }
  DitLength := Round((1.2 / CodeSpeed) * 1000.0);
  FOR I := 1 TO Length(PlainText) DO IF PlainText[I] = '*' THEN
    BlendNextTwo := TRUE ELSE
    BEGIN
      PlainText[I] := UpCase(PlainText[I]);
      CASE PlainText[I] OF
        'A' : CodeChar := '.-';
        'B' : CodeChar := '-...';
        'C' : CodeChar := '-.-.';
        'D' : CodeChar := '-..';
        'E' : CodeChar := '.';
        'F' : CodeChar := '..-.';
        'G' : CodeChar := '--.';
        'H' : CodeChar := '....';
        'I' : CodeChar := '..';
        'J' : CodeChar := '.---';
        'K' : CodeChar := '-.-';
        'L' : CodeChar := '.-..';
        'M' : CodeChar := '--';
        'N' : CodeChar := '-.';
        'O' : CodeChar := '---';
        'P' : CodeChar := '.--.';
        'Q' : CodeChar := '--.-';
        'R' : CodeChar := '.-.';
        'S' : CodeChar := '...';
        'T' : CodeChar := '-';
        'U' : CodeChar := '..-';
        'V' : CodeChar := '...-';
        'W' : CodeChar := '.--';
        'X' : CodeChar := '-..-';
        'Y' : CodeChar := '-.--';
        'Z' : CodeChar := '--..';
        '1' : CodeChar := '.----';
        '2' : CodeChar := '..---';
        '3' : CodeChar := '...--';
        '4' : CodeChar := '....-';
        '5' : CodeChar := '.....';
        '6' : CodeChar := '-....';
        '7' : CodeChar := '--...';
        '8' : CodeChar := '---..';
        '9' : CodeChar := '----.';
        '0' : CodeChar := '-----';
        '?' : CodeChar := '..--..';
        '.' : CodeChar := '.-.-.-';
        ',' : CodeChar := '--..--';
        '/' : CodeChar := '-..-.';
        '$' : CodeChar := '...-..-';
        '-' : CodeChar := '-....-';
       ELSE CodeChar := ''
      END; {CASE}
```

```
        Morse(CodeChar);
        IF NOT BlendNextTwo THEN Delay(DitLength * 2);
        BlendNextTwo := FALSE
      END;
  END;
```

SendMorse works best at code speeds between 15 and 25 WPM and gets dicey below 10 or above 35. For practice at lower speeds it would make sense to modify the code to insert additional dead time between characters and keep the characters elements themselves at 10 WPM or faster.

The short program **MorseTest** demonstrates the use of **SendMorse**.

```
PROGRAM MorseTest;

USES Crt;

TYPE
  String80  = String[80];

{$I SENDMORS.SRC}

BEGIN
  ClrScr;
  SendMorse('CQCQCQ DE KI6RA *SK',850,15);
END.
```

17

Units for Separate Compilation

This chapter might be subtitled, "In Units There is Strength." Why compile the whole shebang when you only need to compile the piece you're working on?

We've been asking this question for years. Pascal naturally separates a program into logical chunks (or it does when you don't fight the spirit of the language). Good program design calls for relatively independent modules which may be compiled separately, and then linked together in one quick, final step before testing the completed program.

This is called "separate compilation." Separate compilation has never been part of the Pascal language definition, but time has shown that it's very hard to manage large projects without it. Turbo Pascal 3.0 and earlier versions did not allow any kind of separate compilation. Turbo Pascal 4.0 and later releases adopt the units paradigm for separate compilation pioneered by UCSD Pascal years ago and used by Turbo Pascal for the Macintosh since its release. This chapter covers the mechanisms by which separate compilation happens in Turbo Pascal.

17.1: THE PACKING LIST METAPHOR

Let's say the UPS man rolls up to your door one day and drops a cardboard box in your lap. You know where it came from (in my case, probably John Meshna & Company, which sells a great many strange and wonderful things), but your memory of what the order contains has gotten a little fuzzy.

You rip open the little plastic slap-on window and pull out the goldenrod sheet of paper marked "packing list." Right in a row is a summary of what's in the box: 2 spur gears, 96 tooth, brass; 1 pillow block ball bearing, ¼"; 1 alarm clock, Navy Surplus (Original cost $900); and 1 tank prism.

Without actually ripping open the box, you know what's in it. If, for example, the packing list read something like "25 polyethelene shower curtains, mauve," I would suspect the UPS man had dropped the wrong box in my lap.

Units are a little like a sealed box with a packing list. Each unit has two primary parts:

1. an *interface* part.
2. an *implementation* part.

The interface part is similar to a packing list. It is an orderly description of what is in the unit, *without* the actual code details of the functions and procedures within the unit. The interface part includes constant, type, and variable definitions, along with the parameter line portions of the functions and procedures within the unit.

This last item may seem a little strange. Consider the following line of code:

```
PROCEDURE FogCheck(InString : String; VAR FogFactor : Integer);
```

This isn't all of the procedure, obviously, but, like it or not, it's all you really need to see of **FogCheck** to be able to make use of the procedure in your own programs. Of course, you need to know the relationship between the input parameter **InString** and the output parameter **FogFactor**, but that's a documentation issue. Knowing *what* procedure **FogCheck** does is an entirely separate matter from knowing *how* it does it. If you know that the foggier the input string is, the higher a value will come back in **FogFactor**, well, that's sufficient.

Where is the rest of **FogCheck**? It's in the *implementation* part of the unit; in other words, it's inside the sealed box. The box contains the substance of the order, the actual goods. The packing list contains a description. That is the critical difference between interface and implementation.

At this point the packing list metaphor begins to break down, because in order to do anything with my brass gears and tank prism I have to rip open the box and take the goods out. A separately compiled unit may remain a sealed box in that you cannot read the details of what lies inside, but the interface part of the unit allows your own programs to hook into and use the contents of the box.

Using Units

Turbo Pascal 4.0 and 5.0 come with several precompiled units full of routines for use in your own programs. This works to your advantage in many ways, not the least of which is that the procedures and functions within those readymade units are already compiled, and do not need to be compiled again every time you compile your own programs, as was the case with Turbo Pascal 3.0. Previously, Turbo Pascal 3.0 supported libraries of routines only as include files, which needed to be compiled every time the main program was compiled. Without having to recompile the units you use, compilation in general goes much more quickly.

How do you use these readymade units? Just like that—you **USE** them:

```
PROGRAM Caveat;

USES Crt;

BEGIN
  ClrScr;                         { In unit Crt }
  GoToXY(12,10);                  { ditto }
  Writeln('Better to light one single candle...'); { NOT in Crt!}
  Writeln('...than to trip on a rake while changing the fuse.');
END.
```

Here, the program **Caveat** contains a new type of Pascal statement: The **USES** statement. **USES** is a reserved word specific to Turbo Pascal 4.0 and later (although some other Pascal implementations define it in roughly the same way, like the P-code UCSD implementation).

Caveat uses a unit called **Crt** that is included with Turbo Pascal. As you might imagine, **Crt** contains routines that deal with screen handling. These include many routines familiar from Turbo Pascal 3.0 that used to be built into the compiler itself. **ClrScr** and **GotoXY** are the most common examples. Such routines were never really part of the Pascal language; but because Turbo Pascal 3.0 did not have any way to perform separate compilation, they had to be built into the compiler to be easily used. There is nothing magical or peculiar about either **ClrScr** or **GotoXY**. They are both ordinary procedures that you could have written yourself.

The same isn't quite true about those other stalwarts, **Read**, **Readln**, **Write**, and **Writeln**. These are *not* procedures in the strictest Pascal sense. If you're sharp you'll know why without being reminded: They can take a variable number of parameters of many different types in any order at all, which is a gross violation of Pascal's rules and regulations regarding procedures. This being the case, they *must* be built into the compiler, because the compiler has to generate different object code to perform each separate call to **Read** or **Write** depending on the number and types of the parameters used. I consider them Pascal statements, the same way **REPEAT/UNTIL** and **WHILE/DO** are statements.

So while most people think of **Writeln** as a CRT-oriented procedure (forgetting, perhaps, its use in writing to text files), it does not "live" in the same unit with all the other CRT-oriented functions and procedures supplies with Turbo Pascal.

The **USES** statement can take any number of unit names, separated by commas. The order you place them in the **USES** statement is not important *unless* subprograms inside one unit reference declarations made inside one of the other units. In that case, in keeping with Pascal's dictum of "define it before you reference it," the callee must be named before the caller. Here's an example of how you can go wrong:

```
PROGRAM Graphical;    { This won't compile! }

USES Graph3,Crt;

BEGIN
END.
```

If you try to compile this program, Turbo Pascal will give you

```
Error 68: Unit not found (CRT)
```

with the cursor under the identifer **Graph3**. Is **Crt** missing? Not at all, and the error message is actually a little misleading. The **Crt** unit is right there on your disk where it always is. The problem here is that the Turbo Pascal 3.0 graphics unit **Graph3** uses the **Crt** unit, and for that reason **Crt** *must* be named *before* **Graph3** in the **USES** statement. If you rearrange the **USES** statement a little, everything will work correctly:

```
USES Crt,Graph3;
```

Where Units Must be Placed on Your Disk

Units physically exist in one of two places: either as separate files with their own names and a .TPU (Turbo Pascal Unit) extension, or as parts of a file called TURBO.TPL. There may be many files on your working disk with the .TPU extension, but with Turbo Pascal 4.0 there can only be one .TPL (Turbo Pascal Library) file, and it currently must be named TURBO.TPL.

TURBO.TPL is the home of Turbo Pascal's runtime library, **System**, along with **Crt**, **DOS**, and several other standard units that always come with the compiler. This is why you will not see separate files on your disk called CRT.TPU, DOS.TPU, and so on. It is a collection of procedures, functions, variables, constants, and other declarations that are used so frequently that they are always loaded into memory and ready to be used by your programs. Note that you never have to name **System** in a **USES** statement. By default, *every* Turbo Pascal program uses **System**.

TURBO.TPL is read into memory whenever you load and run the Turbo Pascal compiler or environment. It thus takes up a certain amount of memory that cannot be used for other things, like data and heap space for programs run under the Turbo Pascal Environment. If you know that you will not be using some of the standard units resident in TURBO.TPL, you can remove them with a standard utility called TPUMOVER.EXE. This will make TURBO.TPL smaller and allow the memory to be used for other things. The use of TPUMOVER is explained of the *Turbo Pascal Owner's Handbook.*

Units that are not part of TURBO.TPL must be kept somewhere the compiler can find them. Unfortunately, you can't place a drive spacifier or a path specifier in the **USES** statement. In other words, these are not legal **USES** statements:

```
USES Crt,DOS,C:RingBuf;           { Won't compile! }

USES Crt,D:\JIVETALK\RINGBUF;     { Won't compile! }
```

The best explanation of this is that unit names are *not* file names, but Pascal identifiers.

Here are the several conditions under which the compiler is able to find your unit files:

- If they're in the current director. In other words, your current directory is C:\JIVETALK, and the compiler is in C:\TURBO. Because C:\TURBO is in your DOS path, you can invoke the compiler from C:\JIVETALK. If the units you wish to use are in C:\JIVETALK, the compiler will find them.
- If you have set up a path to your units in the Directories submenu of the Options menu in the Environment. There is a specifier string for unit directories in that menu that you can fill with one or more paths to directories containing units:

```
Unit directories: C:\TURBO;C:\JIVETALK;C:\IKONYX
```

This is how the prompt would appear on your screen if you had these directories correctly specified to contain units. Notice that the three paths are separated *only* by semicolons. If you leave a space after the semicolon, any following paths will *not* be recognized! When a unit name is called out in **USES** statement, the compiler will search each path until it finds a unit of the correct name. The danger here is having units of the identical name but with different contents in several places on your disk. The *first* one the compiler finds will be the one it uses. If that isn't the one you want, you had better be more specific, as shown in the next item.

- If you precede the unit name with a **$U** compiler command. (Version 4.0 only!) The **$U** command followed by a filename or pathname specifies the file or pathname of the unit that follows the command. For example:

```
USES Crt,Graph, {$U C:\IKONYX\BITBLAST.TPU} Blitter;
```

Here, the **$U** command (note that like all compiler commands it must be placed within comment delimiters) specifies a pathname to a file named BLITTER.TPU that contains the unit named **Blitter**. The **$U** command takes precedence over units in the current directory or in directories specified in the **Options** menu, so that if you have an identical unit in your current directory, the **$U** allows you to select a unit of the same name stored somewhere else on your disk.

Note also that the **$U** command can specify a filename for a unit that is different from the Pascal unit name. Ordinarily, the compiler would look for a unit named **Blitter** in a file named BLITTER.TPU. However, you can specify any name at all in a **$U** command:

```
USES Crt,Graph, {$U C:\IKONYX\BLITTER.TPU} Blitter;
```

In this case, the Pascal unit named **Blitter** is actually stored in a DOS file named BITBLAST.TPU. The name does not have to be the same, but in every case the *extension* must be .TPU or Turbo Pascal will not recognize the file as a unit.

Referencing Identical Identifiers in Different Units

It is perfectly legal to have identical identifiers within two units and use both units from the same program. In other words, you could have a unit called **CustomCRT** that contained a procedure called **ClrScr**, which is the same name as the familiar screen-clearing routine found in standard unit **Crt**:

```
USES DOS,Crt,CustomCrt;
```

A program could use both units as shown above and no error message would be generated. Which **ClrScr** would be actually incorporated into the program?

With no more information than the procedure name to go on, Turbo Pascal will link the *last* procedure named **ClrScr** that it finds in scanning the units named in the

USES statement. In the example above, it would scan **CustomCrt** after scanning **Crt**, and thus it would link the custom-written **ClrScr** routine into the program.

You could exchange the unit names **Crt** and **CustomCrt** in the **USES** statement, and the compiler would then link the standard **ClrScr** routine into your program:

```
USES DOS,CustomCrt,Crt;
```

However, if you arrange the **USES** statement like this, nothing in **CustomCrt** can use any of the many useful routines in **Crt**. This may not be an issue, but it does limit your options. There is a better way.

You can specify the name of the unit that contains an identifier *when you use the identifier.* The notation should be familiar to you from working with Pascal record types, and works in a very similar fashion:

```
PROGRAM WeirdTextStuff;

USES DOS,Crt,CustomCrt;

BEGIN
  Crt.ClrScr;              { Clears the visible PC text screen }
  ClrScr;                  { Clears the other text screens too }

  .  .  .

END.
```

This is often called *dotting.* In the same way that you can have two different record types with identical field names, you can have two or more units containing identical identifiers, and there will be no conflict. You simply choose the one you want by prefixing it with the unit name and a period character at each invocation. The default identifier in cases where no unit name precedes the reference is the first one found in scanning the units in the **USES** statement.

The resemblance to record references ends there. There is no **WITH** statement feature regarding unit names.

17.2: UNIT SYNTAX

In its simplest form, a unit source code file is very much like a Pascal program source code file without a program body. (A unit may have an optional "program body" called the *initialization section,* as I'll explain a little later on.) Typically, units contain procedures and functions, and often other declarations like constants, types, and variables.

A minimal unit with nothing inside it looks like this:

```
UNIT Skeleton;

INTERFACE

IMPLEMENTATION

END.
```

There are several immediate departures from the expected here. The reserved words **INTERFACE** and **IMPLEMENTATION** are *not* statements, and therefore are not followed by semicolons. Like the reserved words **BEGIN** and **END**, they serve to set off groups of statements that belong together.

Also, there is an **END** but no **BEGIN**. Most units do not have a program body. The **BEGIN** is optional, and may be used if the standalone **END** makes you uncomfortable. The **BEGIN** is required if you intend to add an initialization section to the unit.

Fleshing out our unit a little bit will bring out the differences between the interface and implementation parts:

```
UNIT Skeleton;

INTERFACE

USES DOS, Crt;


TYPE
  MyType = ItsDefinition;

VAR
  MyVar : MyType;


PROCEDURE MyProc(MyParm : MyType);

FUNCTION MyFunc(I,Y : Integer) : Char;


IMPLEMENTATION

VAR
  PrivateVar : MyType;


PROCEDURE MyProc(MyParm : MyType);

VAR Q,X : Integer;
```

```
BEGIN
END;

FUNCTION MyFunc(I,Y : Integer) : Char;

BEGIN
END;


END.
```

You should note that the **INTERFACE** keyword must come *before* the **USES** statement, if any. *Nothing*, in fact, may come between the unit name and the **INTERFACE** reserved word.

One type and a variable of that type are defined in the interface section. Both of these definitions are "visible" to any program or unit that uses **Skeleton**. A function and a procedure are also defined in the interface part of **Skeleton**, and like **MyType** and **MyVar** are "visible" to any unit or program that uses **Skeleton**.

Now look down to the implementation section, where the bodies of the function and procedure are given. Notice that a variable name **PrivateVar** is defined in the implementation section. As its name implies, **PrivateVar** is known *only* within the implementation section of the unit. No other program or unit can reference the identifier **PrivateVar** or in any other way know that it exists.

PrivateVar can, however, be accessed by any of the procedures or functions defined within the unit. So, in a sense, the *effects* of **PrivateVar** can be "felt" by outside programs or units that use the subprograms that have access to **PrivateVar**, but the variable itself remains "invisible" to anything outside the unit in which it is defined.

Why is this important? Like so many of Pascal's structural limitations, it is done to minimize the possibility of *sneak paths* occurring between routines when those paths are not desired. Such sneak paths make possible bugs of a truly insidious nature.

17.3: THE INITIALIZATION SECTION OF A UNIT

Structurally, units are very much like Pascal programs without program bodies. The important parts are the definitions of constants, types, variables, and subprograms. A unit may in fact have a "program body," that is, statements between the optional **BEGIN** reserved word and the required **END.** reserved word and period. Any statements in the body of a unit make up that unit's *initialization section*, and they are run *before* the main program body runs.

In a program that uses several units, the initialization section (if there is one) for each one of those units will execute before the main program body begins executing.

The order in which the unit initialization sections run is the same order that the units are named in the **USES** statement. For example:

```
USES Crt,BoxStuff,MenuStuff;
```

When the program that contains this statement is run, the initialization section for **Crt** executes first, followed by the initialization section for **BoxStuff**. There is no initialization section for the unit **MenuStuff**, so as soon as the initialization section of **BoxStuff** finishes executing, the main program body begins executing.

Of what advantage is an initialization section? There are many, although most of them are relatively advanced concepts that you may not need to use until you have come up to speed in Pascal programming in general.

Most simply, an initialization section can initialize global variables declared in the unit to some desired initial value. This relieves the program itself of the responsibility, and avoids the possibility that the programmer will forget to add initialization code to the beginning of his program that uses the unit. Also, in situations where a vendor sells a unit as a separate product, the initialization section guarantees that any globals that need to be initialized *will* be initialized so that the unit (which may not be fully understood by the programmer who uses it) will work correctly without depending on the programmer.

A very simple example involves the **MakeBox** procedure described in Section 14.1. If you remember, **MakeBox** uses a record of type **GrafRec** that contains characters for drawing boxes on the text screen. The record needs to be filled with these characters somehow. In the **BoxTest** program (also given in Section 14.1) that calls **MakeBox**, a procedure called **DefineChars** must be called to fill the **GrafChars** record with the appropriate characters. If that procedure isn't ever called, the string fields in the **GrafChars** record will be undefined and probably contain random garbage characters.

Putting **MakeBox**, the **GrafRec** record, and **DefineChars** together in a unit solve that problem nicely, and provide nearly the simplest useful example of a unit's initialization section in action.

```
{--------------------------------------------------------------}
{                         BoxStuff                             }
{                                                              }
{           Demonstration unit -- draws text boxes             }
{                                                              }
{                             by Jeff Duntemann                }
{                             Turbo Pascal V5.0                }
{                             Last update 7/13/88              }
{                                                              }
{                                                              }
{                                                              }
{--------------------------------------------------------------}


UNIT BoxStuff;


```

```
INTERFACE


USES Crt;     { For GotoXY }

TYPE
  GrafRec = RECORD
              ULCorner,
              URCorner,
              LLCorner,
              LRCorner,
              HBar,
              VBar,
              LineCross,
              TDown,
              TUp,
              TRight,
              TLeft : String[4]
            END;

VAR
  GrafChars : GrafRec;  { Contains box-drawing strings for MakeBox.}
                        { Any program or unit that USES BoxStuff   }
                        { can access the GrafChars variable just   }
                        { as though it had been defined within the }
                        { USEing program or unit.                  }


{<<<< MakeBox >>>>}
{ This is all that the "outside world" really needs to see of the }
{ MakeBox procedure.  *How* it happens is irrelevant to using it. }

PROCEDURE MakeBox(X,Y,Width,Height : Integer;
                  GrafChars        : GrafRec);



IMPLEMENTATION


{<<<< DefineChars >>>>}
{ This procedure is called from the initialization part of the    }
{ unit, and fills the GrafChars record with the box characters.   }
{ Note that it is private to the BoxStuff unit, and *cannot* be   }
{ called from outside the unit.  Note that because it is not part }
{ of the INTERFACE, the full parameter list *must* be given here. }

PROCEDURE DefineChars(VAR GrafChars : GrafRec);

BEGIN
  WITH GrafChars DO
    BEGIN
      ULCorner  := Chr(201);
      URCorner  := Chr(187);
      LLCorner  := Chr(200);
      LRCorner  := Chr(188);
      HBar      := Chr(205);
      VBar      := Chr(186);
      LineCross := Chr(206);
      TDown     := Chr(203);
      TUp       := Chr(202);
      TRight    := Chr(185);
      TLeft     := Chr(204)
    END
END;



{ <<<<MakeBox>>>> }
{ Note here that the parameter line does not have to be repeated. }
{ (We gave the full parameter list definition in the INTERFACE.)  }
{ But since it does no harm, you might as well re-state the       }
{ parameter list.  That makes it easier to read the full source   }
{ for MakeBox. }

PROCEDURE MakeBox(X,Y,Width,Height : Integer;
                  GrafChars        : GrafRec);

VAR
  I,J : Integer;

BEGIN
  IF X < 0 THEN X := (80-Width) DIV 2;    { Negative X centers box }
  WITH GrafChars DO
    BEGIN                                 { Draw top line }
      GotoXY(X,Y); Write(ULCorner);
      FOR I := 3 TO Width DO Write(HBar);
      Write(URCorner);
                                          { Draw bottom line }
      GotoXY(X,(Y+Height)-1); Write(LLCorner);
      FOR I := 3 TO Width DO Write(HBar);
      Write(LRCorner);
                                          { Draw sides }
      FOR I := 1 TO Height-2 DO
        BEGIN
          GotoXY(X,Y+I); Write(VBar);
          GotoXY((X+Width)-1,Y+I); Write(VBar)
        END
    END
END;


{------------------------------------------------------------------}
{ <<<< BOXSTUFF INITIALIZATION SECTION >>>>                        }
{ The initialization section executes before the main block of any }
{ Pascal program that USES BoxStuff.  Here, the initialization     }
{ section fills a record variable called GrafChars with box-drawing }
{ characters, so that the variable is initialized and ready to use }
{ as soon as the main block begins executing.  The programmer does }
{ not have to know ANYTHING about procedure DefineChars.           }
{------------------------------------------------------------------}

BEGIN
  DefineChars(GrafChars);
END.
```

The unit **BoxStuff** contains the record definition for the **GrafRec** type and a public variable **GrafChars** of that type. Both definitions are placed in the interface section, which allows other programs and units to reference them simply by **USING** the **BoxStuff** unit.

The procedure header and parameter list for **MakeBox** are also placed in the interface section, allowing programmers to know exactly how to call **MakeBox**.

That, however, is all there is in the interface section of the unit. The implementation section of the unit contains two procedures: **MakeBox** itself, including all of the "works" inside of it that are not visible in the interface section; and **DefineChars**, which loads the box-drawing characters into **GradChars**.

Notice something about **DefineChars**: It isn't anywhere in the interface section of the unit. The procedure header isn't there anywhere. **DefineChars** is defined completely and solely within the implementation section of **BoxStuff**, and is therefore private to **BoxStuff**. This means that only references from within the implementation section of **BoxStuff** are allowed. The outside world has no way even to know that a procedure called **DefineChars** exists.

Now, look at the very end of unit **BoxStuff**. Between the **BEGIN** and **END**. is a call to **DefineChars**. This call is the only statement in the initialization section of **BoxStuff**. All it does is invoke **DefineChars** to load the box-drawing characters into the **GrafChars** record.

This call is made automatically, before the main program block of any program that uses **BoxStuff**. Nothing has to be done to "start it up;" the Turbo Pascal runtime code takes care of it. When any program using **BoxStuff** begins running, the **GrafChars** variable will be initialized and ready to use from the first statement.

Putting all of the machinery connected with making text boxes in a unit makes the **BoxTest** program a great deal smaller and simpler. Program **BoxTest2** is functionally identical to the **BoxTest** program given in Chapter 14. By using the **BoxStuff** unit, however, it hides all the messy details of setting up a **GrafRec** and making boxes with it. The result is a cleaner, easier-to-read program.

```
{--------------------------------------------------------------}
{                          BoxTest2                            }
{                                                              }
{   Separate compilation demo program -- USES BOXSTUFF.TPU     }
{                                                              }
{                             by Jeff Duntemann                }
{                             Turbo Pascal V5.0                }
{                             Last update 7/14/88              }
{                                                              }
{                                                              }
{                                                              }
{--------------------------------------------------------------}

PROGRAM BoxTest;

USES CRT,BoxStuff;

TYPE
```

```
    String80  = String[80];

  VAR
    X,Y          : Integer;
    Width,Height : Integer;

  BEGIN
    Randomize;                   { Seed the pseudorandom number generator }
    ClrScr;                      { Clear the entire screen }
    WHILE NOT KeyPressed DO      { Draw boxes until a key is pressed }
      BEGIN
        X := Random(72);         { Get a Random X/Y for UL Corner of box }
        Y := Random(21);
        REPEAT Width := Random(80-72) UNTIL Width > 1;  { Get Random Height & }
        REPEAT Height := Random(25-Y) UNTIL Height > 1; { Width to fit on CRT }
        MakeBox(X,Y,Width,Height,GrafChars);            { and draw it! }
      END
  END.
```

The Initialization Section as Watchdog

Another, somewhat more advanced use of the initialization section of a unit is to act as a watchdog to be sure certain conditions are just right before proceeding to execute the program as a whole. If something isn't right, the initialization section can call a halt to the whole program right then and there.

A fairly simple example of this notion concerns the installable device driver used with virtually all mouse pointing devices sold today. Programs that intend to use the mouse need to have the mouse installed in the computer or no pointer will appear and the elaborate pull-down menus often used in conjunction with a mouse will be worthless.

A program that depends on having a mouse installed should always check to see if the mouse driver is in fact loaded and available. This is not difficult, and it is an ideal task for the initialization section of a unit providing high-level language interface to the mouse driver.

The standard Microsoft mouse driver is called by way of an 8086 software interrupt. I'll be discussing software interrupts in some detail in Section 20.2. It's an advanced concept, but it amounts to this: The 8086 CPU supports a table of 256 pointers in low memory. These pointers may be set to point to routines loaded anywhere in memory. The routines may then be called by executing an instruction that transfers control to whatever code that the given pointer in the pointer table points to. The caller does not need to know where in memory the code is; it only needs to know what interrupt number points to the code in question.

The Microsoft mouse driver loads in memory and then arranges for the pointer (more commonly called a "vector") at position 51 in the interrupt vector table to point to it. Programs that wish to use the mouse need only call interrupt 51 and pass it information in the 8086 machine registers.

Now, the question arises: How do we tell if the mouse driver has been installed somewhere in system memory? The answer is to look at interrupt vector 51 in the table. If the vector is **NIL** (that is, if all 32 bits set to 0), then the driver is not installed.

Or, if there is an address in the vector, but that address points to a certain instruction called an IRET (Interrupt RETurn), then the mouse driver has not been installed. Any vector at position 51 in the table that is neither **NIL** nor pointing to an IRET can be assumed to be pointing to a loaded mouse driver.

There are two negative possibilities here because not all computers set up the interrupt vector table the same when you power-up your machine. Some fill the vector table with zeroes, some place a pointer to an IRET in all unused vector positions. It depends on the manufacturer of the BIOS. IBM's PC BIOS places a pointer to an IRET in vector 51, but many of the Far East BIOSes I have tried simply zero-fill the table before setting up the essential vectors.

There is a procedure in Turbo Pascal's standard unit **DOS** that fetches any given interrupt vector from the vector table. By fetching the interrupt 51 vector through the **GetIntVec** procedure, and then testing the pointer returned for a **NIL** value or an IRET byte at its target address, the unit's initialization section can decide whether or not the mouse driver is installed. If the driver is not there, the initialization section code halts the program with an error message before the main program block ever gets control.

The following unit provides a number of functions for using the mouse from your Turbo Pascal programs. Notice that the procedure **MouseCall** is in the implementation section. **MouseCall** is the only procedure in the unit that actually calls interrupt 51. All the other procedures and functions are higher level *wrappers* that handle register manipulation invisibly, allowing you to use the mouse functions without being concerned about mouse function numbers, register values, and so on. **MouseCall** could in fact be moved into the interface section of the unit, and probably should for the reason that revisions of the Microsoft standard mouse driver will certainly add new mouse functions that aren't covered by the higher level functions listed in the interface section. By having the low level procedure **MouseCall** available you can create your own high level calls for any future additions to the mouse function call repertoire.

But keeping **MouseCall** as a procedure private to unit **Mouse** completely hides the gritty details of mouse manipulation from the outside world, which to a certain extent is what units are for.

We'll be making use of the **Mouse** unit later on in the book.

```
{--------------------------------------------------------------}
{             TURBO PASCAL 4.0 MOUSE INTERFACE UNIT            }
{                                                              }
{                                 by Jeff Duntemann            }
{                                 Turbo Pascal V4.0            }
{                                 Last Updated 10/26/87        }
{                                                              }
{                                                              }
{                                                              }
{--------------------------------------------------------------}

UNIT Mouse;

INTERFACE

USES DOS;

```

```

VAR
  ButtonCount : Integer;          { Number of buttons on mouse }


FUNCTION IsLogitechMouse : Boolean;          { Looks at driver }

PROCEDURE ResetMouse;        { Standard Mouse function call 0 }

PROCEDURE PointerOn;                                       { 1 }

PROCEDURE PointerOff;                                      { 2 }

PROCEDURE PollMouse(VAR X,Y : Word;
                    VAR Left,Center,Right : Boolean);      { 3 }

PROCEDURE PointerToXY(X,Y : Word);                         { 4 }

PROCEDURE SetColumnRange(High,Low : Word);                 { 7 }

PROCEDURE SetRowRange(High,Low : Word);                    { 8 }

{PROCEDURE MouseCall(VAR M1,M2,M3,M4 : Word); }



{-----------------------------------------------------------}
{                  MOUSE UNIT IMPLEMENTATION                 }
{                                                           }
{ Note that I choose to re-assert the parameter lists in the }
{ implementation.  It causes no harm and makes this easier  }
{ to read, so why not?  Also note that here, the low-level  }
{ mouse-call procedure Mouser is private to the unit and    }
{ can't be called from outside the unit.  If you wish to do  }
{ more exotic things with the mouse driver (like turn the   }
{ worthless light pen emulation on or off) you should move  }
{ Mouser into the interface to make it accessible.          }
{-----------------------------------------------------------}

IMPLEMENTATION


VAR
  M1,M2,M3,M4 : Word;



PROCEDURE MouseCall(VAR M1,M2,M3,M4 : Word);

VAR
  Regs : Registers;

BEGIN
  WITH Regs DO
    BEGIN
      AX := M1; BX := M2; CX := M3; DX := M4
    END;
  Intr(51,Regs);
  WITH Regs DO
    BEGIN
```

```
      M1 := AX; M2 := BX; M3 := CX; M4 := DX
    END
END;


FUNCTION NumberOfMouseButtons : Integer;

BEGIN
  M1 := 0;  { Must reset mouse to count buttons! }
  MouseCall(M1,M2,M3,M4);
  NumberOfMouseButtons := M2
END;



FUNCTION MouseIsInstalled : Boolean;

TYPE
  BytePtr = ^Byte;

VAR
  TestVector : BytePtr;

BEGIN
  GetIntVec(51,Pointer(TestVector));
  { $CF is the binary opcode for the IRET instruction; }
  { in many BIOSes, the startup code puts IRETs into   }
  { most unused bectors. }
  IF (TestVector = NIL) OR (TestVector^ = $CF) THEN
    MouseIsInstalled := False
  ELSE
    MouseIsInstalled := True
END;


{-------------------------------------------------------------}
{      PROCEDURES ABOVE THIS BAR ARE PRIVATE TO THIS UNIT      }
{-------------------------------------------------------------}



FUNCTION IsLogitechMouse : Boolean;

TYPE
  Signature = ARRAY[0..13] OF Char;
  SigPtr = ^Signature;

CONST LogitechSig : Signature = 'LOGITECH MOUSE';

VAR
  TestVector : SigPtr;
  L          : LongInt;

BEGIN
  GetIntVec(51,Pointer(TestVector));
  LongInt(TestVector) := LongInt(TestVector) + 16;
  IF TestVector^ = LogitechSig THEN
    IsLogitechMouse := True
  ELSE
    IsLogitechMouse := False
```

```
END;



PROCEDURE ResetMouse;

BEGIN
  M1 := 0;
  MouseCall(M1,M2,M3,M4);
END;


PROCEDURE PointerOn;

BEGIN
  M1 := 1;
  MouseCall(M1,M2,M3,M4)
END;


PROCEDURE PointerOff;

BEGIN
  M1 := 2;
  MouseCall(M1,M2,M3,M4)
END;


PROCEDURE PollMouse(VAR X,Y : Word; VAR Left,Center,Right : Boolean);

BEGIN
  M1 := 3;                { Perform mouse function call 3 }
  MouseCall(M1,M2,M3,M4);
  X := M3; Y := M4;       { Return mouse pointer X,Y position }
  IF (M2 AND $01) = $01 THEN Left := True ELSE Left := False;
  IF (M2 AND $02) = $02 THEN Right := True ELSE Right := False;
  IF (M2 AND $04) = $04 THEN Center := True ELSE Center := False;
END;


PROCEDURE PointerToXY(X,Y : Word);

BEGIN
  M1 := 4;
  M3 := X; M4 := Y;
  MouseCall(M1,M2,M3,M4)
END;


PROCEDURE SetColumnRange(High,Low : Word);

BEGIN
  M1 := 7;
  M3 := Low;
  M4 := High;
  MouseCall(M1,M2,M3,M4)
END;


PROCEDURE SetRowRange(High,Low : Word);

BEGIN
```

```
    M1 := 8;
    M3 := Low;
    M4 := High;
    MouseCall(M1,M2,M3,M4)
  END;


{---------------------------------------------------------}
{                   INITIALIZATION SECTION                }
{                                                         }
{  Function MouseIsInstalled goes out and checks the interrupt }
{  51 vector--if it is either NIL (zeroed) or points to an     }
{  IRET, the mouse is assumed NOT to be installed.  Note that  }
{  a vector table full of garbage may be taken to mean the     }
{  mouse driver is there.  Use the VECTORS utility to check    }
{  the state of your vector table after cold boot.  Some Asian }
{  schlock BIOSes may not initialize the table properly and    }
{  cause a false reading on testing for an installed driver.   }
{---------------------------------------------------------}


BEGIN
  IF NOT MouseIsInstalled THEN
    BEGIN
      Writeln('>>>ERROR:  Mouse driver not detected.  Aborting to DOS.');
      HALT(1)
    END;
  ButtonCount := NumberOfMouseButtons
END.  {Mouse}
```

17.4: UNIT EXIT PROCEDURES

Pascal programs that use units don't necessarily begin at the top of the main program block. If any of the units it uses have initialization sections, the program actually begins with the initializations section of the first unit named in the **USES** statement.

The flipside is also true: A program that uses units doesn't necessarily end after the final **END**. of the main program block. An alter ego of the initialization section is available under Turbo Pascal that executes *after* the main program block finishes running. This is the exit procedure.

Exit procedures are trickier to understand than initialization sections, because they involve pointers to procedures, which is nonstandard, low level stuff new even to Turbo Pascal. You might wish to come back to this section after digesting some of the pointer and address functions explained in Section 23.2.

Exit procedures exist primarily to undo what an initialization section may do: Set the machine up in some application-specific fashion. If the initialization section sets up one or more interrupt vectors to point to interrupt service routines contained within the application, something had better restore those vectors to what they were before the application began running.

Exit procedures may also be used simply to "clean house" and ensure that whatever needs doing before returning control to DOS gets done. This might include closing all files, erasing unneeded temporary files, or purging sensitive data from RAM-based buffers.

An exit procedure is an ordinary procedure without parameters. It may not be a function, nor an INLINE nor interrupt procedure. There is no reserved word or other qualifier in Turbo Pascal that brands it as an exit procedure. What makes it an exit procedure is the way it is called, and that only.

The Exit Procedure Chain

Every unit may have an exit procedure, as may the main program. Exit procedures are called after the main program finishes executing, by completing its tasks normally, or by encountering a **Halt** statement, or by falling prey to a runtime error. When a program ceases execution, any existing exit procedures are executed in the following order: First the main program's exit procedure runs, then the unit exit procedures in *reverse* order as the units were declared in the **USES** statement. In other words, given this **USES** statement:

```
USES Unit1,Unit2,Unit3;
```

the main program's exit procedure will run first, followed by **Unit3**'s, then **Unit2**'s and finally **Unit1**'s.

The manner in which exit procedures are called is interesting and a little tricky. The Turbo Pascal runtime library maintains a generic pointer called **ExitProc**. Ordinarily this pointer is set to **Nil**; however, a unit or the main program can set **ExitProc** to point to a procedure designated as an exit procedure. Then, when the runtime library takes control back from the main program and prepares to return to DOS, it tests **ExitProc** and transfers control to the address **ExitProc** contains if **ExitProc** is not equal to **Nil**.

This works simply and well if there is only one exit procedure in a given program. However, with a little attention to details, the main program and every unit can have an exit procedure, and all may make use of the same exit procedure pointer, **ExitProc**.

This process is a rough one to digest without some visual aid. Figure 17.1 can be read two ways: Reading from the bottom to the top, it shows the sequence in which an exit procedure chain is built. Read from the top to the bottom, it shows the sequence in which exit procedures in a chain are called, and the chain dismantled in the process.

Figure 17.1 shows the main program and three units it uses, as would happen in a program with the following **USES** statement:

```
PROGRAM Main;
USES Unit1, Unit2, Unit3;
```

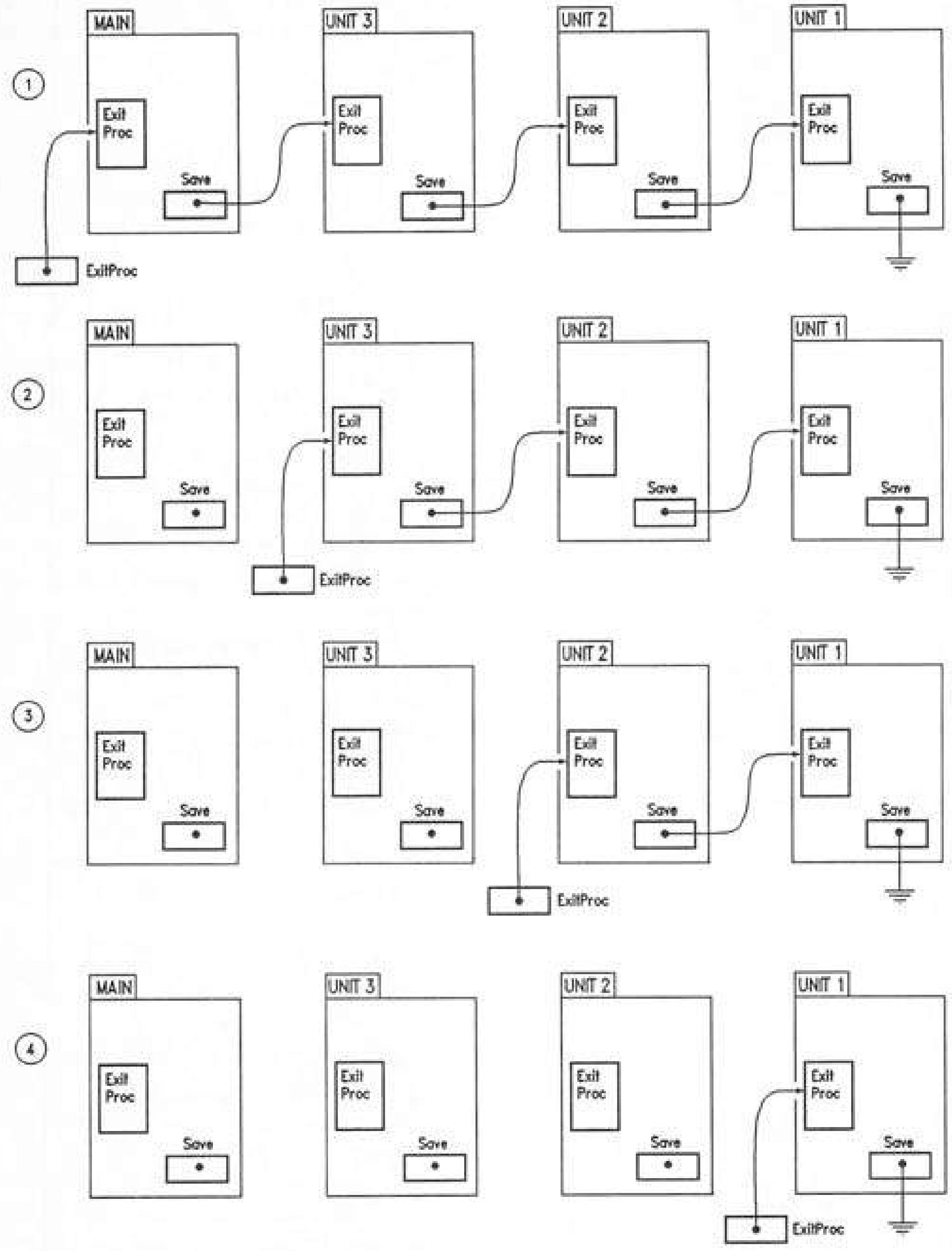
Each unit and the main program have a procedure defined that is to serve as an exit procedure. Also, the main program and each unit has a generic pointer defined which for this example we'll call **Save**.

```
VAR Save : Pointer;
```

Save is local to each unit and can therefore have the same name as both the main program's **Save** and the **Saves** belonging to the other units. As we mentioned above, the

Figure 17.1

The Exit Procedure Chain



predefined generic pointer **ExitProc** is ordinarily set to **Nil** by the Turbo Pascal runtime code.

Creating an exit procedure chain is begun in the initialization section of the first unit that executes prior to **Main**, which in our example is **Unit1**. **Unit1** saves the *current* value of **ExitProc** in its **Save** pointer:

```
Save := ExitProc;
```

Since the initial value of **ExitProc** is **Nil**, **Save** now contains the value **Nil**. **Unit1** next assigns the address of its exit procedure to the pointer **ExitProc**:

```
ExitProc := @Unit1ExitProcedure;
```

This is the situation as shown in row 4 of Figure 17.1. The @ operator means "address of" and it returns the address of its operand, which in this case is the procedure **Unit1ExitProcedure**.

Now, the truly clever thing about the process is that the *next* unit performs *exactly* the same two tasks in its initialization section: It assigns the current value of **ExitProc** to its own **Save** pointer, and then assigns the value of its exit procedure to **ExitProc**. This is the situation shown in row 3 of Figure 17.1. Whereas **ExitProc** had been pointing to **Unit1**'s exit procedure, **Unit2**'s **Save** pointer points now to **Unit1**'s exit procedure, and **ExitProc** now points to **Unit2**'s exit procedure.

For the next step, **Unit3** does exactly the same thing: It places the current value of **ExitProc** into its own **Save** pointer, and assigns the address of its own exit procedure to **ExitProc**. We then have the situation shown in row 2 of Figure 17.1. Finally, in the first few lines of the main program, **Main** continues the process and does what the units did: It puts the current value of **ExitProc** into its **Save** pointer, and puts the address of its own exit procedure into **ExitProc**. The chain is then complete, as shown in the top row of Figure 17.1.

It is crucial that you understand one thing: There is *nothing* about the order of execution built into the units that form the chain of exit procedures. You could change the order of the **USES** statement to:

```
USES Unit2, Unit1, Unit3;
```

and the process would happen the same way, save that **Unit2** would be the first unit in the chain, followed by **Unit1** and finally **Unit3**. Furthermore, none of the units know how many other units are in the chain or if there *are* any other units in the chain.

When **Main** ceases running, whether by ending normally, through a **Halt** statement, or through a runtime error, the entire chain of exit procedures is executed. The order of execution is the reverse of the order in which the units constructed the chain. The process can be followed by reading Figure 17.1 from the top row down. It is a little more subtle than the process of building the chain.

The Turbo Pascal runtime library begins the process by testing **ExitProc** to see if it holds a value of **Nil**. If not, it transfers control to the address contained in **ExitProc** via a **CALL** instruction. In our example, the first exit procedure to execute will be the one belonging to **Main**. This is the situation in the top row of Figure 17.1.

Just before calling the address contained in **ExitProc**, the runtime library moves the address into registers and sets **ExitProc** to **Nil**. This is important in ensuring that the chain is executed in sequence, as we'll see in a moment. Because the exit procedure took control via a **CALL** instruction and not a **JMP** instruction, *control returns to the runtime library when the exit procedure finishes executing.* The runtime library then tests the value of **ExitProc** against **Nil** again; and it calls the address in **ExitProc** if **ExitProc** is not equal to **Nil**.

If this sounds like an infinite loop, you're right. It is. However, the exit procedure has one task to perform before it returns control to the runtime library, and that is to assign the value of its **Save** pointer to **ExitProc**:

```
ExitProc := Save;
```

Save, if you recall (or can see from Figure 17.1), contains the address of the *next* exit procedure in the chain. Therefore, when the first exit procedure returns control to the runtime library, **ExitProc** contains the address of the next exit procedure in the chain. The runtime library tests **ExitProc** against **Nil** again, and calls the second exit procedure in the chain. This is the situation in row 2 of Figure 17.1.

Just before returning control to the runtime library, the exit procedure for **Unit3** assigns its own **Save** pointer to **ExitProc**, bringing us to the situation shown in row 3 of Figure 17.1. Once again, the runtime library calls the address in **ExitProc** and the execution of the chain continues.

Now, suppose for a moment that the first exit procedure had *not* set **ExitProc** to **Nil** before returning to the runtime library? In that case, **ExitProc** would still contain the address of the first exit procedure. The runtime library might then call the same exit procedure over and over again in an infinite loop. However, there is a safety mechanism built into the runtime library: Before calling an exit procedure, the runtime library sets **ExitProc** to **Nil**. That way, if the exit procedure fails to set **ExitProc** to the address of the next exit procedure in the chain, **ExitProc** will contain **Nil** on return to the runtime library, and the runtime library will stop executing the chain.

All things come to an end. Consider what happens when the last exit procedure, that belonging to **Unit1**, terminates. It assigns the value of its **Save** pointer to **ExitProc**. But because it was the first exit procedure in the chain, the value it stored in its **Save** pointer was the default value placed in **ExitProc** by the runtime library: **Nil**. Therefore, when control returns to the runtime library, **ExitProc** will contain **Nil**, and the runtime library will consider the chain finished. The runtime library will then complete its own housecleaning and return to DOS.

This business of exit procedure chains is complicated, but it is reliable and provides a level of safety unknown in earlier versions of Turbo Pascal and in most other language compilers.

I have written four simple source code files that perform the sequence of events we have been describing in the last few pages. They do nothing more than create an exit procedure chain and then exit it, finally returning to DOS. Follow the code through until it all makes sense to you.

```
UNIT Unit1;

INTERFACE

IMPLEMENTATION

VAR Save : Pointer;

{$F+} PROCEDURE Unit1ExitProcedure; {$F-}

BEGIN
  Writeln('Unit #1 -- Exit procedure...');
  ExitProc := Save
END;

BEGIN
  Save := ExitProc;
  ExitProc := @Unit1ExitProcedure;
  Writeln('Unit #1 -- Initialization procedure...');
END.
```

```
UNIT Unit2;

INTERFACE

IMPLEMENTATION

VAR Save : Pointer;

{$F+} PROCEDURE Unit2ExitProcedure; {$F-}

BEGIN
  Writeln('Unit #2 -- Exit procedure...');
  ExitProc := Save
END;

BEGIN
  Save := ExitProc;
  ExitProc := @Unit2ExitProcedure;
  Writeln('Unit #2 -- Initialization procedure...');
END.
```

```
UNIT Unit3;

INTERFACE

IMPLEMENTATION
```

```

VAR Save : Pointer;

{$F+} PROCEDURE Unit3ExitProcedure; {$F-}

BEGIN
  Writeln('Unit #3 -- Exit procedure...');
  ExitProc := Save
END;

BEGIN
  Save := ExitProc;
  ExitProc := @Unit3ExitProcedure;
  Writeln('Unit #3 -- Initialization procedure...');
END.
```

```
{--------------------------------------------------------------}
{                            Main                              }
{                                                              }
{      Exit procedure chaining demonstration program           }
{                                                              }
{                              by Jeff Duntemann               }
{                              Turbo Pascal V4.0               }
{                              Last update 7/1/88              }
{                                                              }
{                                                              }
{                                                              }
{--------------------------------------------------------------}

PROGRAM Main;

USES Unit1,Unit2,Unit3;

BEGIN
  Writeln('Main program execution begins here.');
  Writeln('Main program execution ends here.');
END.
```

Once you understand how they work, compile and link them together into an executable file and run them. If what you see when they run seems natural to you, then you have digested all there is to know about Turbo Pascal's exit procedures.

17.5: UNITS AS OVERLAYS (VERSION 5.0)

Turbo Pascal 1.0 did not have overlays. Turbo Pascal 2.0 and 3.0 had overlays. Turbo Pascal 4.0 does not. Overlays are back with release 5.0, and let us hope that this time they're here to stay. With overlays, your program can get a great deal larger than the DOS 640K memory limit, as long as your disk system is large enough to hold the program's component parts.

The idea behind overlays is just that: You cut a program into parts, and arrange it so that several of the parts share the same place in memory. When the main program begins running, an initial overlay is loaded into a slot in memory. When the second overlay is required, it is loaded into memory into the same slot as the first overlay, "overlaying" the first overlay. Later on, when a third, fourth, or fifth overlay is required, it is simply loaded on top of whatever overlay had previously occupied the overlay area in memory. In this way, you can have megabytes of code that all run handily in 256K RAM or even less. You simply need to carve those megabytes of code up into a sufficient number of overlays.

Overlays from a Height

Turbo Pascal 5.0's overlay system is based on units. Any number of units may be specified as overlays, which means that each will occupy the same region of memory when loaded. The overlaid units, in a sense, take turns at executing in memory, and on loading each unit overwrites the unit existing in memory before it.

All units specified as overlays are linked to a separate file, with the same name as the main program but with an extension of .OVR. In other words, MYPROG.PAS, if it contains overlaid units, will compile to MYPROG.EXE and MYPROG.OVR.

Unlike Turbo Pascal's earlier overlay system, there is only one overlay area in memory. This is called the *overlay buffer*, and it is actually allocated on the heap before the main program takes control, in effect bumping the heap up by the size of the overlay buffer (see Figure 23.1 for a detailed memory map of a Turbo Pascal program at runtime). So there is no need to be concerned about multiple overlay areas. You simply specify that a unit is to be overlaid, and the compiler takes care of the rest.

The overlay buffer is initially made as large as it must be to contain the largest overlaid unit. The overlay buffer can be made larger if desired, as we'll explain later.

If the runtime library detects expanded (EMS) memory in the system at runtime, it will allow the .OVR file to be loaded into EMS RAM. From that point on, all accesses will be made from EMS RAM to the overlay buffer at RAM speed, without any further access to the disk-based copy of the .OVR file. Because expanded RAM is only slightly slower than DOS RAM, the end result of using EMS RAM is to defeat the DOS 640K barrier. By using EMS, your programs can be completely memory-based and as large as they need to be, up to the practical limits of the runtime library.

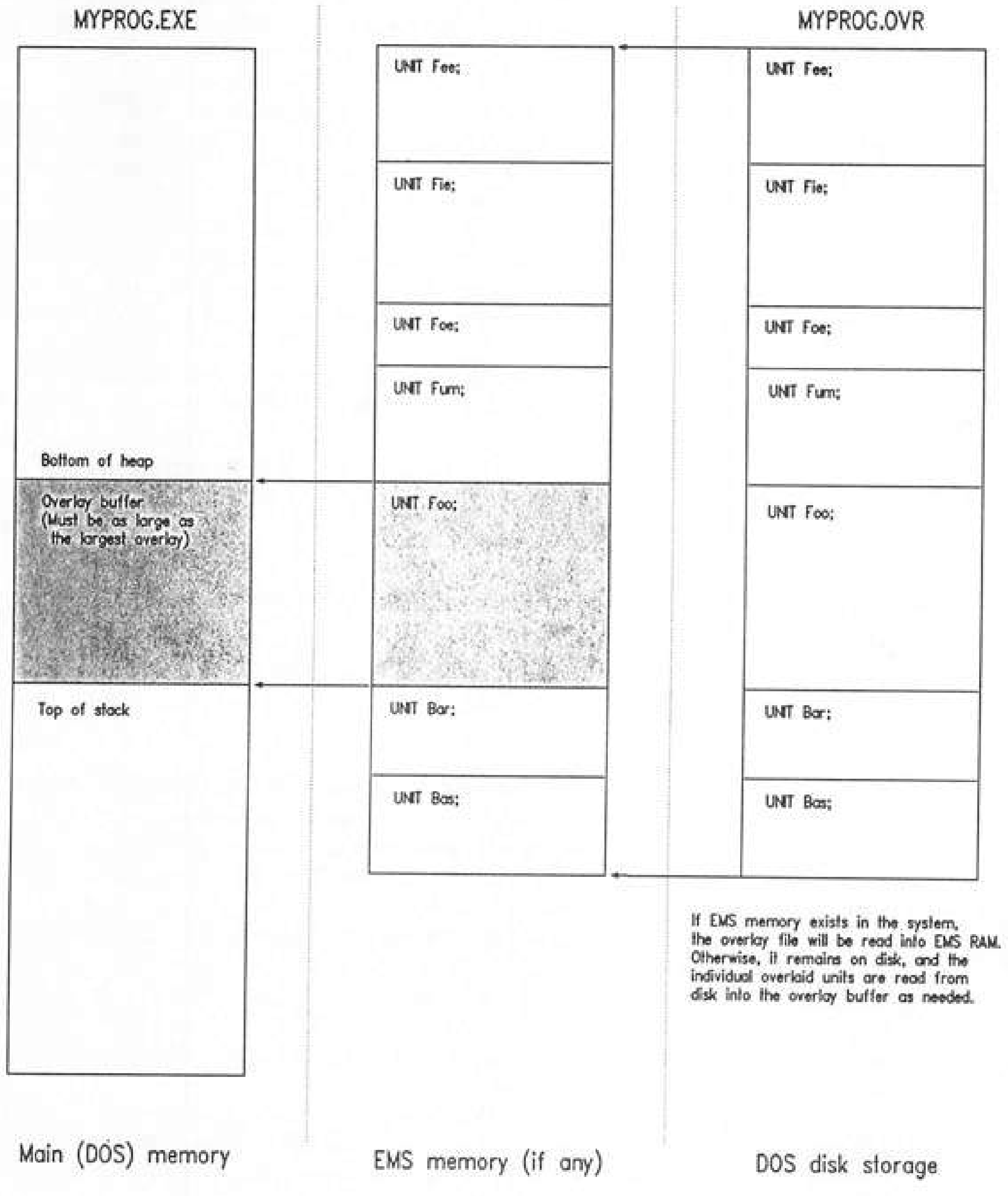
Turbo Pascal's overlay system is laid out in Figure 17.2.

Setting Up an Overlaid Application

Turbo Pascal provides an overlay manager that handles access to overlays, in a unit called **Overlay**. If you intend to use overlays, you must include **Overlay** in your **USES** statement, before *all* of the overlaid units. As a safety precaution, you might get in the habit of specifying **Overlay** first any time you specify it at all.

Figure 17.2

Overlay Files and Buffers


MYPROG.EXE
MYPROG.OVR
UNIT Fee;
UNIT Fie;
UNIT Foe;
UNIT Fum;
UNIT Foo;
UNIT Bar;
UNIT Bas;
Bottom of heap
Overlay buffer (Must be as large as the largest overlay)
Top of stack
If EMS memory exists in the system, the overlay file will be read into EMS RAM. Otherwise, it remains on disk, and the individual overlaid units are read from disk into the overlay buffer as needed.
Main (DOS) memory
EMS memory (if any)
DOS disk storage

A unit is specified as overlaid through a new compiler directive: **$O**. (Compiler directives are discussed in detail in Section 27.3.) The directive has two forms: A toggle form and a parameter form. The toggle form consists of the **$O** directive followed by either a plus or a minus symbol. The parameter form consists of the **$O** directive followed by its parameter, which in this case is the name of a unit to be overlaid. Both forms of the **$O** directive appear within comment delimiters.

The two forms of the **$O** directive go in two different places, but actually in the service of the same purpose: To define a unit as an overlay. The toggle form goes in the unit source code file itself, to "turn on" overlay support code generation within that unit. The parameter form is placed in the main program, to tell the compiler which units are to be treated as overlays. You are given two chances to specify overlays because you may want the option of making a unit an overlay in one program or not an overlay in another, as required. By placing the **$O+** directive in the unit source file, you *enable* its use as an overlay, but you don't *require* its use as an overlay. It can still be linked normally if desired.

The parameter form of the toggle, by contrast, tells the main program that the named unit is to be treated as an overlay. Such **$O** directives must be placed *after* the **USES** statement:

```
PROGRAM JiveTalk;

{$F+}

USES Overlay,DOS,CRT,CircBuff,XMODEM,Kermit,Packet1K,Parser;

{$O XMODEM}
{$O Kermit}
{$O Packet1K}
```

This example overlays some but not all of its units. **Overlay** is first in the **USES** statement. There is no necessary order in placing the **$O** directives.

A common error is to place the **$O** parameter directives in the main program with the names of units that do *not* have a **$O+** directive in them. Without **$O+** in the unit source code file, *that unit cannot be overlaid.*

There was definitely a method in choosing which units in program **JiveTalk** were to be made overlays. The three units contain code for three file transfer protocols. Since the program services only a single communications session at a time, only one of the three protocols will be active at any one time. Furthermore, once a file transfer has been initiated, it will carry through without requiring the use of any protocol other than the one currently operating.

On the other hand, while the **DOS** unit could be overlaid, to do so would be insane, because nearly every other unit in the program calls the **DOS** unit. If another overlay called **DOS**, the two would be waltzing in and out of memory by turns in a process we call "thrashing," which would be slow indeed.

This means don't put "toolkit" units out as overlays. Instead, design major subsystems as giant overlaid units, and arrange it so that none of the subsystems call elements of other subsystems. For example, in an accounting application put the accounts payable module out as an overlaid unit, the accounts receivable module out as another unit, payroll as a third unit, and so on; but keep screen control and menuing routines out of the overlay scheme. In short, overlay the *highest* level routines, and leave low-level and utility routines in memory, to be available to all and sundry.

One additional point to note in the example declarations above is the presence of a **$F+** compiler directive immediately after the **PROGRAM** statement. One of the requirements of a program containing overlays is that all calls must be far calls. Place a **$F+** directive at the start of any program using overlays. Failing to do so may cause a subprogram call to walk off the edge of the Earth, taking your program session with it.

Overlay Management

The overlay manager needs to be initialized at runtime before any of the routines in the overlay file are called. The **Overlay** unit provides a procedure to accomplish this:

```
PROCEDURE OverInit(OverlayFileName : String);
```

The name of the DOS file containing the overlays is passed in **OverlayFileName**. This name can be a full pathname, including drive unit specifier and subdirectory path. If no drive unit is specified, the overlay manager searches the current drive, and if no path is specified, the overlay manager searches the current directory. If the file is not found in the current directory, the overlay manager will then search the directory containing the .EXE file (providing you're using DOS 3.X). Then it will search directories listed in the DOS PATH environment variable.

If an error occurs during overlay manager initialization, an error is reported in the predefined variable **OvrResult**. All of the routines in unit **Overlay** report errors in **OvrResult**. Each of the possible values is represented by a predefined constant in **Overlay**. A table of the constants, their values, and their meanings is given in Table 17.1.

OvrResult should be tested immediately after calling **OvrInit**. If an error is returned by **OvrInit**, it will typically be **OvrNotFound**, meaning the overlay manager cannot locate the specified overlay file. The generic **OvrError** code may also be returned, indicating an error not covered by any of the other codes. If anything other than **OvrOK** is returned in **OvrResult** after a call to **OvrInit**, the overlay manager will not be installed, and if you attempt to call a routine in an overlaid unit, runtime error 208 will occur, halting your program.

Something to note about **OvrResult**: Unlike **IOResult**, which it resembles functionally, **OvrResult** is not set to zero immediately after being read, but instead remains unchanged until the next call to a routine in the **Overlay** unit. You needn't copy **Ovr-**

Table 17.1
Predefined Constants in Unit Overlay

Code	*Constant*	*Meaning*
0	OvrOK	A-OK
−1	OvrError	"None of the above" error code
−2	OvrNotFound	Specified overlay file not found
−3	OvrNoMemory	Not enough heap for overlay buffer
−4	OvrIOError	I/O error loading overlay file
−5	OvrNoEMSDriver	EMS driver not installed
−6	OvrNoEMSMemory	Not enough EMS memory for overlay file

Result into a variable for safekeeping if you don't want to, unless for some reason you need to keep the error status of one call to an overlay manager routine during or after a call to another.

Turbo Pascal's overlay manager has the ability to detect the presence of EMS memory at runtime. If EMS memory is detected, the overlay manager can use it; if not, no harm is done, and the overlay file can remain on disk. However, in order for this detection to work, you need to explicitly call an overlay manager routine:

```
PROCEDURE OvrInitEMS;
```

Note that there are no parameters, and that it is *not* a function. Also keep in mind that **OvrInitEMS** in no way *replaces* **OvrInit**. Once you have successfully initialized the overlay manager, you should call **OvrInitEMS** to see if any EMS is available. If enough EMS memory is available to hold the overlay file, the overlay file will be loaded into EMS memory. It's as simple as that, and totally automatic:

```
OverlayName := 'JIVETALK.OVR';
OvrInit(OverlayName);
IF OvrResult <> OvrOK THEN
CASE OvrResult OF
  OvrNotFound :  { Overlay file not found on disk }
    BEGIN
      Writeln('Overlay file ',OverlayName,' not found.');
      Writeln('Please reinstall JiveTalk and run again.');
      Halt(OvrResult)
    END;
  OvrIOError :   { I/O error loading overlay file }
    BEGIN
      Writeln('I/O error reading overlay file ',OverlayName,'.');
      Writeln('The file may be corrupted.  Please re-install');
      Writeln('JiveTalk from the master disk.');
      Halt(OvrResult)
    END;
  OvrNoMemory :  { Not enough heap to allocate overlay buffer }
    BEGIN
      Writeln('There is not enough memory in your computer');
```

```
        Writeln('to run Jivetalk.  You need at least 384K RAM.');
        Writeln('Without additional memory, Jivetalk cannot run.');
        Halt(OvrResult)
      END;
    ELSE           { Miscellaneous overlay system error }
      BEGIN
        Writeln('Jivetalk overlay system error.  Please reboot');
        Writeln('your system and run again.  If error persists,);
        Writeln('re-install JiveTalk from the master disk.');
        Halt(OvrResult)
      END;
  END;  { CASE }

  OvrInitEMS;      { Detect and use EMS RAM if available }
  IF OvrResult <> OvrOK THEN
    IF OvrIOError THEN
      BEGIN
        Writeln('I/O error reading overlay file ',OverlayName,'.');
        Writeln('The file may be corrupted.  Please re-install');
        Writeln('JiveTalk from the master disk.');
        Halt(OvrResult)
      END;
```

This code fragment from a communications program illustrates an unavoidable truth about commercial quality applications: Most of their bulk is devoted to detecting and dealing gracefully with either system or user errors. The days when a commercial application can simply roll over on its back and die are long past. Your users deserve better. Try to be ready for all errors, and at very least exit to DOS with some sort of "last words."

Note that no special action is taken on an **OvrResult** value indicating that there is no EMS driver installed or no EMS memory in the system. These are normal situations, because many or most people do not have EMS memory in their system.

Setting the Overlay Buffer Size

When **OvrInit** runs, it allocates a buffer on the heap large enough to contain the largest overlaid unit (see Figure 17.2.). If you can afford to give the overlay manager more heap memory than that, it will speed overall system performance by allowing the overlay manager to keep more than one unit in memory at a time. An algorithm that keeps an eye on which overlays are called most frequently allows the overlay manager to decide which overlays to maintain in memory and which to leave on disk. You, however, must tell the overlay manager how large it may make the overlay buffer, using another procedure from the **Overlay** unit:

```
PROCEDURE OvrSetBuf(BufferSize : LongInt);
```

Here, **BufferSize** specifies the total desired size of the overlay buffer, in bytes. This amount of memory is allocated on the heap. Two conditions are necessary for a call to

OvrSetBuf to succeed: First, the heap must be *empty*. If *any* dynamic variables have been allocated on the heap before you call **OvrSetBuf**, the call will fail and the **OvrError** error code will be returned in **OvrResult**. Second, there must be enough memory on the heap to satisfy the request. If there is insufficient heap memory available, an **OvrNoMemory** error is returned in **OvrResult**.

Keep in mind that by the time you call **OvrSetBuf**, it is assumed that you have already called **OvrInit**, and that **OvrInit** has been able to successfully allocate enough memory on the heap to contain the largest single overlaid unit. Therefore, if a call to **OvrSetBuf** fails, your program can continue operating. It will simply have to be content with an overlay buffer as large as, but no larger than, the largest overlay.

Querying the Overlay Buffer Size

A companion routine to **OvrSetBuf** allows you to query the current size of the overlay buffer:

```
FUNCTION OvrGetBuf : LongInt;
```

Quite simply, the size of the overlay buffer in bytes is returned by **OvrGetBuf**. **OvrGetBuf** and **OvrSetBuf** allow your programs to do some smart memory allocation by looping through a series of attempts—perhaps starting by asking for enough heap space to hold the entire overlay file until a memory allocation value small enough to succeed is found.

Overlay Cautions

There are a number of things to keep in mind when designing a program that is to be divided into overlays:

- *Do not attempt to overlay the **Overlay** unit.* The compiler will allow it, but the resulting program will not run.
- *Do not overlay any unit that contains an interrupt handler.* An interrupt handler must be in memory at all times, because when interrupts are triggered, there is no mechanism to intercept control long enough for the overlay manager to load an overlay containing the handler into memory. Again, the compiler will not forbid you to put interrupt handlers into overlays, but if an interrupt is triggered while that overlay is not in memory, your program will crash hard.
- *Avoid designing initialization sections into overlaid units.* There is no real danger in doing this, but you must remember that on program startup, every unit initialization section is run. If you have seventeen overlays, each with an initialization section, each of the seventeen will be loaded into memory just long enough for its initialization section to run. This means a period of constant disk access at the start of your program that will only make the program look bad to impatient users.

18 Controlling Your CRT

Such a simple thing as controlling the placement of information on a CRT screen has been a thorn in the side of computer programmers as long as there have been computers. Each manufacturer of computers and terminals has considered his or her own set of screen control codes the best set possible, and the predictable result has been chaos. Most vendors of computer languages have simply given up and left CRT control as an exercise for the programmer.

Turbo Pascal is one of the few native-code Pascal compilers for microcomputers that attempts to define its own CRT interface standard. Its screen control procedures are not part of ISO Standard Pascal, but *someone* must set a standard if we are to have both portable Pascal code and full-screen Pascal applications. Other Pascal vendors have begun to implement Turbo's screen control procedures in their own compilers, and my feeling is that the standard has firmly taken root.

I make a fuss about CRT control because it is *important.* The manner in which a program interfaces with its user is, to my way of thinking, the most crucial attribute of that program. The days of the "Mystical Order of the Computer Priesthood" are over. A computer program that is hard to use will be left unused. In designing the concept of a computer program, lay out the face that the program presents to the user *first.* Your primary allegiance as a programmer is to the needs of your user.

18.1: THE CRT UNIT'S TEST-ORIENTED FUNCTIONS AND PROCEDURES

Starting with version 4.0, Turbo Pascal accomplishes its text-mode CRT control through a number of functions, procedures, and predefined variables contained in a unit called **Crt**. These routines are a superset of the text support routines contained in version 3.0 of the compiler. The extended graphics procedures from Turbo Pascal 3.0 are now present in a unit called **Graph3**. These are part of Turbo Pascal 4.0 and 5.0 *only* for compatibility purposes; for new development, I recommend starting fresh with the much-superior Borland Graphics Interface (BGI) contained in the unit **Graph**. The BGI is described fully in Chapter 22.

Using the Crt Unit

Like any unit, **Crt** is used by specifying it in your program's **USES** statement:

```
PROGRAM JiveTalk;

  USES CRT,DOS,CommStuf;
```

You can do minimal screen output without using **Crt**, but the only screen routines actually available outside of **Crt** are **Read** and **Readln** for screen input, plus **Write** and **Writeln** for screen output. These routines are defined in the **System** unit and are always available. However, they only perform "glass teletype" screen output. You have no control over where the text appears on the screen. It appears at the beginning of the first clear line and moves down the screen until the screen is full. At that point, the screen scrolls upward one line, much like a piece of paper in an old teletype printer.

DOS Video Versus BIOS Video Versus Direct Video

This is not necessarily a bad thing in all applications. Using **Readln** and **Writeln** without **Crt** uses DOS for screen input and output. DOS is slow, but it allows you to *redirect* screen input and output to text files. For example, a file search routine such as **Locate** (described in Section 20.9) doesn't need to position the cursor or clear the screen. By eschewing **Crt**, **Locate** allows you to redirect its output to a text file, essentially taking a snapshot of the information it gathers. This redirection is done with the > operator on the DOS command line:

```
C:\>LOCATE *.BAK > BAKLIST.TXT
```

Running **Locate** in this way displays nothing at all to the screen, but instead writes a file named BAKLIST.TXT containing its information:

```
 4919  09/20/87   9:23a   \HACKS\POORBACK.BAK
10875  10/24/87   9:34p   \HACKS\VECTORS.BAK
 2454  10/24/87   8:53p   \HACKS\HEXTEST.BAK
29256  08/08/87   9:12p   \LOCATE2.BAK
```

Less useful is *input* redirection, which takes a text file of strings and feeds them one by one to the **Readln** statements within your program:

```
C:\>GENSWEEP < SWEEPCMD.TXT
```

Here, a program called **GenSweep** takes its commands from a text file called SWEEP-CMD.TXT. **GenSweep** then works automatically, rather than requesting input from you through the screen. This feature is handy for creating utility programs that must be used either interactively from the command line or else automatically from within batch files.

The final advantage to using glass teletype I/O through DOS is that your programs will work correctly on any machine that can run DOS. Even machines like the Victor 9000, which are *not* screen-compatible with the IBM PC, will run programs like **Locate** that do not use the **Crt** unit. This limits your options to a certain kind of command-line oriented utility, but you should be aware of the option if you ever feel like building replacements for the utility programs that come with DOS.

When you use the **Crt** unit, you have two additional choices: BIOS screen I/O and *direct* screen I/O. The default is direct screen I/O, which is both the fastest method and the most specific to the IBM PC video architecture.

The DirectVideo Variable

Your choice is specified through a predefined Boolean variable called **DirectVideo**. When **DirectVideo** is **True**, screen output is sent directly to the video buffer, without any BIOS involvement. When **DirectVideo** is **False**, all screen I/O is performed through BIOS calls (Essentially through software interrupt $10). The default value is **True**.

DirectVideo is set to **True** in the initialization section of the **CRT** unit, and after any call to the **TextMode** procedure. If you wish to use the BIOS to perform screen I/O, you must set **DirectVideo** to **False** *before* sending anything to the screen, and after *every* invocation of **TextMode**.

Is there any compelling reason to work through the BIOS? Not really. Any computer that is BIOS-compatible for text video almost certainly is video buffer compatible as well. If you have trouble with screen output, try using BIOS output by setting **DirectVideo** to false. By and large, BIOS calls are extremely wasteful of CPU time, and you are much better off using direct video I/O.

Dealing with CGA Video "Snow"

On IBM's original CGA (Color Graphics Adapter) and many early CGA compatible video boards, writing directly to screen memory generates a scattering of random "snow" across the screen during the instant it takes to write a character directly into memory. Newer video boards like the EGA and VGA and nearly all recent CGA lookalikes use dual-ported memory and avoid snow. If you use Turbo Pascal's direct video I/O and notice snow on your screen during screen updates, you can suppress the snow with a predefined Boolean variable named **CheckSnow**.

CheckSnow defaults to **False**. When set to **True**, it forces the Turbo Pascal runtime code to write characters to the screen buffer *only* during horizontal retrace periods. In this way, the snow can only happen when the monitor's electron gun is turned off, and the snow will not be visible. *However*, this interval is rather small, and forcing all video to be displayed during horizontal retrace will slow your screen displays down enormously. If at all possible, buy a newer video board that is immune to snow, and leave **CheckSnow** set to **False**.

Screens Versus Windows

As I'll explain below, Turbo Pascal's CRT display routines operate relative to the current screen window. Ordinarily, Turbo Pascal considers the current window to be the entire physical screen. There are times, however, when it would be very convenient to treat separate areas of the screen as though each were a separate screen. Writing an application that does two or more separate things comes to mind, as in a split-screen editor or a communications program that monitors both COM1: and COM2: at the same time.

The **Window** procedure exists for this purpose. It is predeclared this way:

```
PROCEDURE Window(X1,Y1,X2,Y2 : Integer);
```

Parameters **X1** and **Y1** define the column and row of the upper left corner of the window to be defined, and **X2** and **Y2** define the lower right corner. Once a window is defined with **Window**, it remains in force until another window is defined, or until you execute the **TextMode** statement.

On the IBM PC, all screen commands operate with respect to the current window, which is either the full screen default or the last window set via **Window**. In other words, if you define a window by

```
Window(10,10,70,20);
ClrScr;
```

only the screen in the rectangular region defined by the given coordinates will be cleared. Surrounding areas of the screen will not be affected.

Similarly, using **GotoXY** within a window *always* uses the upper left corner of the window as the coordinate origin. In other words, with the window mentioned above set at 10,10 and 70,20, performing a **GotoXY(1,1)**; will position the cursor at what appears to be screen location 10,10. However, because the window was set, Turbo Pascal treats 10,10 as though it were the upper-left corner of the entire screen. Physical screen position 10,10 becomes the logical screen position 1,1.

Window makes it possible to selectively clear rectangular areas of the IBM PC screen very quickly—more quickly, in fact, than you can follow by eye, and certainly more quickly than writing lines full of spaces. Using this procedure:

```
{<<<< ClearRegion >>>>}
{                                         }
{                                         }
{                                         }

PROCEDURE ClearRegion(X1,Y1,X2,Y2 : Integer);

BEGIN
  Window(X1,Y1,X2,Y2);
  ClrScr;
  Window(1,1,80,25)
END;
```

you can clear any rectangular area of the screen.

The coordinates of the current window are available in a pair of variables, **WindMin** and **WindMax**, both of type **Word**. **WindMin** contains the upper-left X,Y coordinates, with X in the low-order byte and Y in the high-order byte. Similarly, **WindMax** contains the lower-right X,Y coordinates of the current window, with X in the low-order byte and Y in the high-order byte. To separate them, you need to use the built-in functions **Lo** and **Hi** (see Section 16.9). The other thing to remember in using **WindMin** and **WindMax** is that they contain the current window coordinates *relative to zero.* In other words, if you have set the current window to 10,10–50,18 **WindMin** will contain 9,9 and *not* 10,10. **WindMax** will contain 49,17 rather than 50,18.

WindMin and **WindMax** are actually a peek at the storage area in which the Turbo Pascal runtime keeps its own low level information on the window, and the runtime thinks of window coordinates as relative to zero. Remember to add 1 to the individual coordinates (and *not* to the entire word variable **WindMin** or **WindMax**!) before you use them.

Screen Control Routines Defined in CRT

The rest of this section describes the routines that actually send characters to the screen. There are other routines in **CRT** that have nothing to do with screen output. These include **Delay**, **Sound**, and **NoSound**. They really belong among Turbo Pascal's standard functions, and I describe them in Section 16.13.

All the routines described below operate *within the current window.* The default window is the full screen, but if you define a smaller window to be the current window, the action of these routines will be limited to within that window.

ClrScr

This command blanks the current window and returns the cursor to the upper left hand corner. If the current window is the entire screen (the default) the entire screen will be blanked. Turbo Pascal considers the coordinates of the upper left corner of the screen to be 1,1, *not* 0,0 as many other programming environments (such as dBase II) do.

The **TextMode** procedure also clears the screen, but it clears the entire physical screen, and not simply the current window.

GotoXY(X,Y)

This command moves the cursor to position **X,Y** in the current window where **X** is the column (counting across from the left) and **Y** is the row (counting down from the top.) **X** and **Y** may be type **Integer**, **Byte**, **Word**, **LongInt**, or **ShortInt** but not any real number type. The upper left corner is position 1,1.

ClrEOL

The **EOL** in this command means *End Of Line.* **ClrEOL** will clear the line containing the cursor from the cursor to the end of the line. The cursor itself does not move. Keep in mind that the line will only be cleared within the current window.

DelLine

The cursor line is deleted, and lines are moved upward to fill in. An empty line is inserted at the bottom of the current window.

InsLine

An empty line is inserted at the cursor line. The line that previously contained the cursor is moved down beneath the inserted line, and all lines further down move down one line. The bottom screen line will be pushed off the bottom of the screen and lost.

LowVideo

LowVideo clears the high-intensity bit in the video attribute byte used in displaying text sent to the screen. (This byte is available to you as the predefined variable **TextAttr**.) In essence, this maps colors 8 to 15 onto colors 0 to 7. The end result is that characters sent to the screen after **LowVideo** is executed will show up as the lower intensity version of the currently selected color. On a monochrome screen, this means the characters will be dimmer than characters normally appear.

NormVideo

What **LowVideo** does, **NormVideo** undoes. Text sent to the screen after **NormVideo** is executed will be displayed using the video attribute that was in force before **LowVideo** was invoked.

18.2: TEXT MODES, COLORS, AND ATTRIBUTES

In the past few years, the IBM PC has established what must be the closest thing to a CRT display standard that now exists in the microcomputer industry. Unlike previous versions of Turbo Pascal, Turbo Pascal 4.0 and later versions are highly specific to the IBM PC and its close compatibles, especially in terms of CRT handling.

In this section we'll discuss text modes, text colors, and monochrome text attributes.

Text Modes

There are several different variations in text mode for the IBM standard display adapters. The **TextMode** command will select from among the variations. Unlike version 3.0, Turbo Pascal 4.0 and later versions will not allow you to call **TextMode** without a parameter to select the current or last text mode selected. Now, the predefined constant **Last** must be used. The other predefined constants that may be used as parameters to **TextMode** are given in the examples below:

```
TextMode(Last);      { Selects current or last mode used }
TextMode(C40);       { Selects 40-column with color enabled }
TextMode(C80);       { Selects 80-column with color enabled }
TextMode(BW40);      { Selects 40-column with color disabled }
TextMode(BW80);      { Selects 80-column with color disabled }
TextMode(Mono);      { Selects 80-column on mono adapter }
```

Disabling color on the color graphics adapter means suppressing the color burst signal on the composite video output of the adapter board. This causes composite monitors and TV sets to display what would otherwise be a color signal in black and white. The color burst signal exists *only* on the composite output. *If you are using an EGA, a VGA, or the RGB outputs of the Color Graphics Adapter, you cannot disable color!*

On the monochrome display adapter, there is, of course, no color available. Neither is there really a 40-column mode as we know it on the color card; if you select the **C40** or **BW40** parameters, text will be displayed on the left half of the screen only, with characters their normal size and width. IBM's monochrome card cannot display the double-width characters used by the color card in 40-column mode. There is no point, then, in using the **BW40** or **BW80** parameters with **TextMode** when you have the monochrome card installed in your computer. Use **Mono** instead.

No matter which display adapter you are using or what variation of text mode you select, **TextMode** *always* clears the screen. **TextMode** also resets the current window to the entire physical screen.

Text Colors and Monochrome Attributes

If you're using one of the video boards that supports color (CGA, EGA, or VGA), you have complete control over both the foreground and background colors for each individual character position on the screen. Furthermore, if you wish, you can make a character of any color blink. Two built-in procedures give you control of text colors: **TextColor** and **TextBackground**.

These same two procedures also work if you have the monochrome display adapter installed, but what they do is a great deal different. The monochrome display does not display colors, but it allows you to set attributes on a character-by-character basis. These attributes include underline, dim, and blink. Setting an attribute is done the same way you would set a text mode color on a color board.

Setting character color or attribute is done by passing a parameter to **TextColor**. The parameter is an integer having a value from 0 through 31:

```
TextColor(4);        { Selects yellow on color card }
```

The meaning of the parameter for color or monochrome is summarized in Table 18.1.

The constants column in the table lists the 16 integer constants predefined by Turbo Pascal for the values 0 through 15. By using these constants you can make your code a little more self-explanatory:

```
TextColor(LightGreen);
```

Table 18.1
BGI Color and Attribute Constants

		Mode of operation	
Value	*Constant*	*Color*	*Monochrome*
0	Black	Black	Off
1	Blue	Blue	Dim underline
2	Green	Green	Off
3	Cyan	Cyan	Dim
4	Red	Red	Dim
5	Magenta	Magenta	Dim
6	Brown	Brown	Dim
7	LightGrey	Light grey	Dim
8	DarkGrey	Dark grey	Off
9	LightBlue	Light blue	Normal Underline
10	LightGreen	Light green	Off
11	LightCyan	Light cyan	Normal
12	LightRed	Light red	Normal
13	LightMagenta	Light magenta	Normal
14	Yellow	Yellow	Normal
15	White	White	Normal
16		Black	Off
17		Blue blink	Dim blink underline
18		Green blink	Off
19		Cyan blink	Dim blink
20		Red blink	Dim blink
21		Magenta blink	Dim blink
22		Brown blink	Dim blink
23		Light grey blink	Dim blink
24		Dark grey blink	Off
25		Light blue blink	Normal blink underline
26		Light green blink	Off
27		Light cyan blink	Normal blink
28		Light red blink	Normal blink
29		Light magenta blink	Normal blink
30		Yellow blink	Normal blink
31		White blink	Normal blink

There is one additional predefined integer constant, **Blink**. **Blink**'s value is 128. If you add **Blink** to a color constant, you will get that color character that blinks:

```
TextColor(Red + Blink);
```

Again, this is only to make your code clearer; you could just as easily have plugged the literal 132 into the parameter and the result would have been the same.

The difference between the normal colors and the light colors is the intensity signal in the color graphics adapter's RGB outputs. Light red is actually red with the intensity signal on, and so on. Some low-cost color monitors do not make use of the intensity signal, and on those monitors your light colors will be identical to their corresponding normal colors.

On monochrome video boards the color values select various combinations of the dim, underline, and blink attributes. To display underlined text, you would invoke **TextColor** with a value that selects the underline attribute, display the text, and then invoke **TextColor** again with the attribute value previously in force. Although there are no predefined constants for the various attributes, nothing would prevent you from defining constants of your own for that purpose:

```
CONST
  Blink        = 27;
  Dim          = 3;
  Normal       = 11;
  Underline    = 9;
  DimBlink     = 19;
  DimUnderline = 1;

TextColor(Underline);        { Select normal underline text }
Writeln('Be sure to back up your data!');
TextColor(Normal);           { Select normal text display }
```

The background color of a character is the color taken on by the rest of the little rectangular cell that contains the character itself. For the background color you have your choice of the seven normal colors 1 through 8 and of course, color 0, black:

```
TextBackground(Blue);
```

As we show here, the predefined color constants may be used for **TextBackground** as for **TextColor**.

On the monochrome adapter, the background color translates to a black background (colors 0 and 2) or a white background (colors 1,3,4,5,6, and 7). To display in inverse video, for example, you must display text in color black on a white background:

```
TextColor(Black);
TextBackground(1);
Writeln('Use inverse video for emphasis.');
TextColor(Normal);
TextBackground(Black);
```

People converting text applications from Turbo Pascal 3.0 or earlier versions should bear in mind that the **GraphBackground** procedure is no longer supported, starting with Turbo Pascal 4.0. An undocumented use of **GraphBackground** was in setting the text border color. You must eliminate all references to **GraphBackground** or your program will not compile. In the EGA and VGA, the text border is so narrow it is not much worth bothering with.

Saving the Cursor Position

The IBM PC implementation of Turbo Pascal includes two functions that return the current X and Y position of the cursor:

```
FUNCTION WhereX : Byte;

FUNCTION WhereY : Byte;
```

WhereX returns the X coordinate of the cursor. **WhereY** returns the Y coordinate. Both return byte values.

What are **WhereX** and **WhereY** good for? One application lies in the saving of entire text screens. "Painting" a screen with **GotoXY** and **Write** statements takes a certain amount of time. If your application needs to switch frequently among a number of different screens, you'll find yourself spending a noticeable amount of time watching the screens regenerate themselves.

It is possible to save a screen after it is drawn by using the **Move** routine (see Section 23.4) to move a copy of the entire 4K text mode video buffer to another location in free RAM. Then you can bring in another screen in a fraction of a second (assuming it had been saved to RAM as well after being painted to the screen or loaded from a screen file). The problem with this is that the cursor position is *not* saved in the video refresh buffer, but elsewhere in system RAM. If you need to save a cursor position along with a screen, you must save it yourself. **WhereX** and **WhereY** allow you to assign the current cursor position to a pair of integer variables. When bringing in a previously saved screen, you can then restore the cursor position from those two variables with **GotoXY**.

18.3: USEFUL SCREEN CONTROL TACTICS

In this section we'll give you some tips on creating effective text displays from Turbo Pascal.

The **LowVideo** procedure provides a way to highlight (or lowlight, if you prefer) text as a way of setting it off from normal text. The problem is that making text dimmer than the bulk of the text on the screen is a backhanded way of calling attention to it. A better method to accent text is to execute **LowVideo** *immediately* at the start of your program so that half brightness is used for the bulk of the text displayed from your program. Then, when you want to highlight something, drop into normal video for a line or two and your text will stand out nicely:

```
NormVideo;
Writeln('Please call your service rep immediately!');
LowVideo;
```

You'll find after a while that you'll be using familiar sequences of statements to do many display chores. For instance, to place a message at a particular place on the screen takes at least two statements:

```
IF ERROR THEN
  BEGIN
    GotoXY(10,24);
    Writeln('>>That file is missing or damaged!')
  END;
```

Highlighting your message with a video attribute (half-brightness or inverse video) adds two more statements to the compound statement:

```
IF ERROR THEN
  BEGIN
    LowVideo;
    GotoXY(10,24);
    Writeln('>>That file is missing or damaged!');
    NormVideo
  END;
```

If you want to center a message you will have to determine its length, subtract its length from 80 (the width of your typical screen) and divide the difference by two. All of this fooling around may be done inside a single text display procedure:

```
{<<<< WriteAt >>>>}
{                                                 }
{                                                 }
{                                                 }

```

```
PROCEDURE WriteAt(X,Y : Integer;
                  Highlite : Boolean;
                  UseCR    : Boolean;
                  TheText  : String);

BEGIN
  IF Y < 0 THEN Y := 12;
  IF X < 0 THEN X := (80-Length(TheText)) DIV 2;
  GotoXY(X,Y);
  IF Highlite THEN LowVideo;
  IF UseCR THEN Writeln(TheText) ELSE Write(TheText);
  NormVideo
END;
```

WriteAt will automatically center a line of text on your screen from side to side and from top to bottom, either or both as required. If you pass **WriteAt** a negative **X** parameter, your text line will be centered from side to side. If you pass it a negative **Y** value, the text line will be placed on line 12, which is centered on a 25 line screen.

In the constant declaration part of your program you should define a few Boolean constants for the use of **WriteAt**:

```
CONST
  Highlite : True;
  Normal   : False;
  CR       : True;
  NoCR     : False;
```

Rather than plugging the nondescript literals **True** and **False** into **WriteAt**'s parameter line, you can use the descriptive constants given above and make the intent of the parameters clear at a glance:

```
IF Error THEN WriteAt(-1,24,Highlite,NoCR,'<<DISK ERROR!>>');
```

This particular invocation of **WriteAt** will place a highlighted error message centered on line 24 of the screen. Since sending a carriage return to the screen after line 24 will scroll the screen up, the **UseCR** parameter is passed a **False** value by Boolean constant **NoCR**.

WriteAt's purpose is actually to hide the details involved in centering a text line on the screen, highlighting it, and placing it at a particular X,Y position. These details add to the bulk of your code and may serve to obscure the real job of the program. When displaying an error message the important thing to understand is why an error message must be displayed at that point, not how one highlights and centers a text line on the screen.

This is also a good place to point out that it is perfectly legal to break up a procedure call into several separate lines. This can make for less cluttered, easier-to-read source code. Pascal, which does not ordinarily respect the line structure of text files, does not care:

```
WriteAt(HPos + 6, YPos + Offset + 2,
        Highlite, NoCR,
        'This is a raid.  I repeat, this is a raid!');
```

The only thing you *can't* break out into more than one line is a text literal. You could not, in the line above, have written:

```
WriteAt(HPos + 6, YPos + Offset + 2,
        Highlite, NoCR,
        'This is a raid.
         I repeat, this is a raid!');
```

Turbo Pascal would stop with Error 8: String constant exceeds line. To pass **WriteAt** a string literal longer than will fit on a single line, you must concatenate two shorter string literals:

```
WriteAt(HPos + 6, YPos + Offset + 2,
        Highlite, NoCR,
        'This is a raid.  I said, son, this is a raid.' +
'  Listen up in there!  Did you hear me?  THIS IS A RAID!');
```

Unless the screen were wider than 80 characters, which is unlikely, the above message would likely "wrap" around to the next line down when displayed on the screen.

A Function for Yes/No Questions

Perhaps the most common questions to occur during dialogs between a computer program and a user are simple yes/no questions. Coding such a question up involves something close to this:

```
Write('>>Do you want to add another record? (Y/N): ');
Read(Ch);
IF NOT (Ch IN ['Y','y']) THEN
  Quit := True ELSE Quit := False;
```

These lines of code, with a different prompt, of course, would have to be repeated for every yes/no question in the program. Everything except the prompt can be bundled into a single function called **Yes**:

```
{<<<< Yes >>>>}
{                                                 }
{                                                 }
{                                                 }

FUNCTION Yes : Boolean;
```

```

VAR
  Ch : Char;

BEGIN
  Read(Ch);
  IF Ch IN ['Y','y'] THEN Yes := True ELSE Yes := False
END;
```

Now you can do this instead:

```
Write('>>Do you want to add another record? (Y/N): ');
IF Yes THEN Quit := True ELSE Quit := False;
```

The details of getting a character from the keyboard and seeing what it is are hidden by function **Yes**. The dialog nature of the interchange remains the important and visible characteristic of this piece of code. It is a cleaner, more readable way of handling yes/no dialogs in your programs.

A Data Entry Routine For Boolean Variables

Lots of different types of data come in pairs. Human beings are male or female, alive or dead. Telephones are TouchTone or pulse. Cars are U.S.-made or foreign-made. Such data can be conveniently expressed as Boolean values:

```
VAR
  TouchDial,Male,Deceased,USMade : Boolean;
```

Displaying and entering Boolean values within a data entry screen involves a certain amount of rigamarole. For example, one way of displaying the current value of a Boolean variable and entering a new value might be this:

```
REPEAT
  Quit := False;
  GotoXY(1,5); Write('>>TouchTone or Pulse dialing? ');
  IF TouchDial THEN Write('TouchTone')
    ELSE Write('Pulse dial');
  Write(' Change? (Y/N): ');
  IF Yes THEN TouchDial := NOT TouchDial ELSE Quit
UNTIL Quit
```

Yes is the yes/no question function described above. A cleaner and more self-explanatory method of getting a Boolean value from the keyboard is the following procedure:

```
{->>>>FlipField<<<<------------------------------------------}
{                                                             }
{ Filename : FLIPFLD.SRC -- Last Modified 7/14/88             }
{                                                             }
{ This routine facilitates entry of "toggle" fields--fields   }
{ that have one of only two different values: Male/Female,    }
{ Citizen/Non-citizen, etc.  You specify an X,Y position for  }
{ the field, a string for each of the True and False case, and }
{ an initial Boolean value that specifies which of the two    }
{ alternatives will be initially displayed.  Pressing ESC     }
{ during entry changes nothing and sets the ESC parm to True. }
{ Whichever state is displayed at the point the user presses  }
{ Return is the Boolean value returned in VAR parm State.     }
{                                                             }
{                                                             }
{                                                             }
{-------------------------------------------------------------}

PROCEDURE FlipField(X,Y         : Integer;
                    VAR State   : Boolean;
                    TrueString  : String30;
                    FalseString : String30;
                    VAR Escape  : Boolean);

VAR Blanker   : String80;
    KeyStroke : 0..255;
    WorkState : Boolean;
    Ch        : Char;



PROCEDURE ShowState(NowState : Boolean);

BEGIN
  GotoXY(X,Y); Write(Blanker);    { Erase the old label }
  IF NowState THEN
    BEGIN
      GotoXY(X,Y);
      Write(TrueString)   { Write TrueString for NowState = True }
    END
  ELSE
    BEGIN
      GotoXY(X,Y);
      Write(FalseString);  { Write FalseString for NowState = False }
    END
END;


BEGIN
  Escape := False; Ch := Chr(0);
  LowVideo;                                { Use highlighting }
  FillChar(Blanker,SizeOf(Blanker),' ');   { Set up Blanker String }
  WorkState:=State;                        { Temporary Boolean }
  IF Length(TrueString)>Length(FalseString) THEN  { Adjust Blanker }
    Blanker[0] := Chr(Length(TrueString)) ELSE    {  String for lengths }
    Blanker[0] := Chr(Length(FalseString));       {  of meaning labels  }
  ShowState(WorkState);                    { Display initial label }
  REPEAT
```

```
      WHILE NOT KeyStat(Ch) DO BEGIN {NULL} END;  { Calls KeyStat... }
      KeyStroke := Ord(Ch);
      IF KeyStroke = 27 THEN Escape := True ELSE
        IF KeyStroke<>13 THEN WorkState := NOT WorkState;
      ShowState(WorkState);
    UNTIL (KeyStroke=13) OR Escape;          { ...until CR or ESC is pressed }
    IF NOT Escape THEN State:=WorkState;     { Update State if CR }
    NormVideo;
    ShowState(State);     { Redisplay State in non-highlighted text }
  END;
```

When you use a Boolean variable in a program, it probably has a distinct and important meaning connected with each of its two states, **True** and **False**. **FlipField** displays the meaning of variable **State** and allows you to flip between its two states by pressing any key that is neither the escape key nor carriage return.

FlipField is passed two text labels in its parameter line: **TrueString** describes the meaning of **State** when **State** is **True**. **FalseString** describes the meaning of **State** when **State** is **False**. For example, if **State** were a Boolean variable called **PatientSex**, **TrueString** might hold the string *Male* and **FalseString** might hold the string *Female*. When **FlipField** begins running, it displays at **X,Y** the label corresponding to the current state of **State** as passed in the parameter line.

FlipField then waits on the keyboard for keypresses. If an ESC is entered, **FlipField** returns without making any change to **State**. If CR is entered, **FlipField** updates **State** to reflect the currently selected meaning and returns. If any other key is pressed, **FlipField** flips to the other meaning for **State** and updates the screen display at **X,Y** to match. Going back to our example, by pressing a key (space bar works well) repeatedly, **FlipField** will rapidly display "Male" and "Female" at the same location until either ESC or CR is pressed.

All the user must do to pick one of two states for a Boolean value is press CR when the meaning he wants is displayed on the screen. All possible keypresses are accounted for, and he doesn't have to type a string at the keyboard with the possibility of misspellings.

18.4: DETERMINING WHICH DISPLAY ADAPTER IS INSTALLED

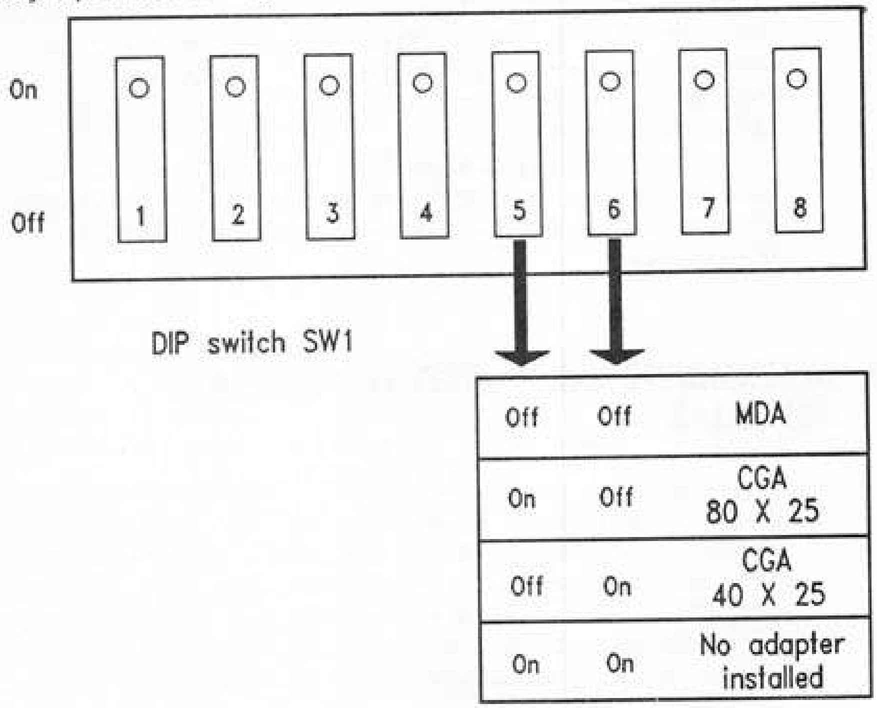
Not counting the shunned and mediocre Professional Graphics Controller and the display board from the defunct PCjr, IBM has released only four distinct display adapters, the CGA, MDA, EGA, and VGA, as of July 1988. Actually, yet another board, the MCGA, exists, but it is a subset variant of the VGA and is nearly identical to the VGA in text modes. A host of third-party display adapters is available, but in general they all try to "look like" one of these IBM boards to software. This mimicry is driven by a need to fall under one of four different DIP switch settings on the PC system board.

Switches 5 and 6 of switch block 1 must be set to reflect the kind of display adapter installed in the system. These switch settings are summarized in Figure 18.1. Switches 5 and 6 are what the PC looks at when it wishes to determine what sort of display it has. It does *not* go out and somehow examine the board to see what is actually present on the bus. This seems to be a risky system on the surface, because there is no promise that a third-party board pretending to be an MDA is actually similar enough to an MDA not to cause trouble. In fact, this risk has perhaps driven third-party display board manufacturers to be a little more careful of how compatible their products actually are with the IBM standard.

It is important to remember that the setting marked "No display adapter installed" means, more properly, the statement from the PC ROM BIOS, "I don't know what adapter is installed—*yet*." It is a way of allowing ROM BIOS to hedge its bets and defer a decision to a better authority.

What authority? Unlike the older CGA and MDA cards, the EGA and VGA contain a ROM BIOS extension in a ROM chip right on the display board. This ROM contains software that is better able to assess the machine environment and decide how to handle video. The EGA or VGA BIOS code takes control during the "ROM

Figure 18.1
Display Dip Switch Settings



Off	Off	MDA
On	Off	CGA 80 X 25
Off	On	CGA 40 X 25
On	On	No adapter installed

scan" period after the motherboard-based ROM BIOS completes its power-on self-test (POST) and initializes the BIOS data areas. The EGA or VGA BIOS then alters the BIOS data areas as it sees fit.

We know where in memory the BIOS data areas are located, but IBM does not promise not to move them in future releases or future additions to the PC family. Therefore, the safe bet (and speed matters little here since it needs to be done only once) is to access this information through the BIOS. PC ROM BIOS function $11 returns 16 bits' worth of information on what equipment the PC thinks it has installed. This information comes from the BIOS data areas, *not* directly from the DIP switches as some people believe. The DIP switches are read *only* at POST time.

This is an important distinction, because the EGA and VGA BIOS overwrite the DIP switch information when they perform tests and initialization on the EGA or VGA board during ROM scan. Recall that the EGA and VGA installation instructions direct the user to set DIP switches 5 and 6 to indicate *no* display adapter installed. The EGA and VGA determine through their own means whether a color or monochrome monitor is attached to the system, and, if a color monitor is attached, the EGA/VGA BIOS sets the BIOS data area as though the DIP switches had been set to reflect a CGA. Likewise, if the EGA or VGA detects a monochrome monitor attached to the system, it sets the BIOS data areas as though the DIP switches had indicated an MDA.

So *don't* make the mistake of assuming that you will get a 0 code back from BIOS function $11 if the installed video display adapter is an EGA or VGA. If an EGA or VGA is installed, BIOS will tell you there is a CGA installed if a color monitor is attached to the system, or BIOS will say an MDA is installed if a monochrome display is attached.

Obviously, BIOS function $11 alone will *not* tell you whether an EGA or VGA is actually installed or not. Remarkably, there is no really straightforward way of detecting the presence of an EGA or a VGA in a system. The method I use has proven reliable for IBM's own EGA and VGA plus a large number of clone boards, and I suspect it will work reliably on any EGA or VGA with a responsibly written ROM BIOS on the board.

The generalized display-board detection method works this way: In a sense, we are detecting BIOS upgrades rather than boards. The VGA BIOS routines for VIDEO contains a new function call, $1A, that returns a code number identifying the nature of the installed display adapters and monitors. You might wonder why the VGA BIOS would bother returning a code for older cards—if the VGA BIOS is present, the adapter must be a VGA, right?

Wrong. In a sense, the VGA BIOS is the new standard for *all* display card BIOSes in the PS/2 world, and there would be nothing to stop IBM from creating a low-cost monochrome display adapter with a BIOS containing the new VIDEO function calls.

So we start by trying to invoke VIDEO function call $1A. If the function call is present, it will return the function call number (here, $1A) in register AL. (This is a general standard for behavior in the PS/2 BIOS—if a function call exists and is called, it will return its function number in AL.) If call $1A happens, we take its say-so for the current display card type. While all display adapter types are covered, in reality you'll only see the VGA or the MCGA (standard with the PS/2 models 25 and 30.)

If we determine (from testing AL against $1A) that no PS/2 BIOS exists in the machine, the next step is to test for an EGA BIOS. Like the PS/2, the EGA ROM BIOS adds a few extensions to the interrupt $10 VIDEO service included in the PC/XT/AT motherboard ROM BIOS. One extenion is service AH=$12, Alternate Function. Service $12 currently includes two subfunctions. We are interested in subfunction BL=$10, Return EGA Information. This subfunction will poll the installed EGA and return 1) the mode in effect, either color or monochrome; 2) the amount of EGA memory installed; 3) the bits from the EGA feature connector; and 4) the settings of the DIP switch on the spine of the EGA.

Consider what might happen if you made this subfunction call on a machine that did *not* contain an EGA. In such machines, VIDEO service AH=$12 is not defined. VIDEO handles such cases in a chivalrous manner and ignores them, restoring all registers and returning to the caller without changing anything.

In a sense, if we ask for EGA information in a machine with an EGA, we get back EGA information in register BX. If we ask for EGA information in a machine *without* an EGA, we get back *exactly* what we passed to VIDEO in BX, which (thank fate!) is not valid EGA information. So the method consists of performing a Return EGA Information call to VIDEO and seeing if BX changes in the process. If BX changes, we have an EGA. If BX stays the same, no EGA is installed.

Finally, if we find neither a VGA nor an EGA, we poll interrupt $11 and see what it tells us. At that point the choice is essentially between a CGA and an MDA.

This series of tests allows us to build a function that returns a value indicating which card is installed in the machine. Ideally, a function should return a distinct value for each kind of display card. While returning a plain numeric value (0, 1, 2, etc.) would be adequate, Pascal allows us to define an enumerated type which will make programs much more self-explanatory:

```
TYPE
  AdapterType = (MDA,CGA,EGAMono,EGAColor,VGAMono,
                 VGAColor,MCGAMono,MCGAColor);
```

The EGA, VGA, and MCGA appear in two incarnations, because they can be configured to look (in text mode, at least) nearly identical to either the CGA or the MDA, including the starting address for the video buffer. The two EGA/VGA/MCGA incarnations are sufficiently different (especially since they cannot be switched from one to another under software control) to consider them, from a software perspective, as two different kinds of display adapter.

QueryAdapterType returns a value of type **AdapterType**, which means that you must define **AdapterType** in any program that uses **QueryAdapterType**, or else build the type and the function into a unit.

```
{->>>>QueryAdapterType<<<<------------------------------------}
{                                                              }
{ Filename : QUERYDSP.SRC -- Last Modified 7/11/88             }
{                                                              }
{ This routine determines the currently installed primary      }
{ display adapter and returns it in the form of a value from   }
{ the enumerated type AdapterType.                             }
{                                                              }
{ AdapterType must be predefined:                              }
{                                                              }
{ AdapterType = (None,MDA,CGA,EGAMono,EGAColor,VGAMono,        }
{                VGAColor,MCGAMono,MCGAColor);                 }
{                                                              }
{                                                              }
{                                                              }
{--------------------------------------------------------------}

FUNCTION QueryAdapterType : AdapterType;

VAR
  Regs : Registers;
  Code : Byte;

BEGIN
  Regs.AH := $1A;  { Attempt to call VGA Identify Adapter Function }
  Regs.AL := $00;  { Must clear AL to 0 ... }
  Intr($10,Regs);
  IF Regs.AL = $1A THEN  { ...so that if $1A comes back in AL...  }
    BEGIN                { ...we know a PS/2 video BIOS is out there. }
      CASE Regs.BL OF    { Code comes back in BL }
        $00 : QueryAdapterType := None;
        $01 : QueryAdapterType := MDA;
        $02 : QueryAdapterType := CGA;
        $04 : QueryAdapterType := EGAColor;
        $05 : QueryAdapterType := EGAMono;
        $07 : QueryAdapterType := VGAMono;
        $08 : QueryAdapterType := VGAColor;
        $0A,$0C : QueryAdapterType := MCGAColor;
        $0B : QueryAdapterType := MCGAMono;
        ELSE QueryAdapterType := CGA
      END { CASE }
    END
  ELSE
  { Next we have to check for the presence of an EGA BIOS: }
    BEGIN
      Regs.AH := $12;       { Select Alternate Function service }
      Regs.BX := $10;       { BL=$10 means return EGA information }
      Intr($10,Regs);       { Call BIOS VIDEO }
      IF Regs.BX <> $10 THEN { BX unchanged means EGA is NOT there...}
        BEGIN
          Regs.AH := $12;   { Once we know Alt Function exists... }
          Regs.BL := $10;   { ...we call it again to see if it's... }
          Intr($10,Regs);   { ...EGA color or EGA monochrome. }
          IF (Regs.BH = 0) THEN QueryAdapterType := EGAColor
            ELSE QueryAdapterType := EGAMono
        END
      ELSE  { Now we know we have an EGA or MDA: }
        BEGIN
          Intr($11,Regs);   { Equipment determination service }
          Code := (Regs.AL AND $30) SHR 4;
          CASE Code of
            1 : QueryAdapterType := CGA;
            2 : QueryAdapterType := CGA;
            3 : QueryAdapterType := MDA
            ELSE QueryAdapterType := CGA
          END { Case }
        END
    END;
END;
```

Character Cell Size and Cursor Manipulation

Before the introduction of the EGA, dealing comprehensively with text video was a much simpler thing. The CGA used an 8 × 8 character cell; the MDA used an 8 × 14 character cell. (For the sake of you purists: Yes, it is a *9* × 14 character cell, but the ninth pixel row doesn't really exist!) The EGA has *both* sizes of fonts in its ROM, as we mentioned above, for both monochrome and color modes. The VGA complicates matters still further by having both the 8 and 14 pixel fonts, plus an 8 × 16 pixel font as well. The MCGA, just to be perverse, has the 8 × 16 pixel font but *not* the 8 × 8 or 8 × 14 fonts.

Knowing what font the EGA or VGA happen to be using at any given time is important mostly because turning the cursor on and off correctly depends on it. The reason is this: It's *easy* to turn the cursor off; a simple call to VIDEO service 1 with bit 5 of CX high will do it.

```
{<<<< CursorOff >>>>}
{                                                             }
{                                                             }
{                                                             }
{ HIGHLY specific to the IBM PC }

PROCEDURE CursorOff;

VAR
  Regs : Registers;

BEGIN
  WITH Regs DO
    BEGIN
      AX := $0100;
      CX := $2000;   { Set CH bit 5 hi to suppress cursor }
    END;
  INTR(16,Regs);
END;
```

Turning the cursor back *on* is another story. The PC's text cursor is defined as a range of scan lines within the character cell, with the top scan line 0, and the bottom scan line either 15, 13, or 7, depending on the font in use at the time. Although any range of scan lines, including the entire character cell, can be set to flash as the cursor, the default text cursor consists of the bottom two scan lines in the character cell. Turning the cursor on again requires passing to VIDEO the starting scan line and ending scan line of our desired cursor. There is no way to "turn the cursor on as it was before we turned it off." VIDEO doesn't "remember" the old cursor settings. They get thrown away when you turn the cursor off. We need to know how many scan lines are in the current character cell before we can turn the default cursor back on correctly.

The function **DeterminePoints** returns the number of scan lines in the font currently in use. (Calling it "**Points**" hearkens back to "point sizes" of printed fonts, although the number of pixels in a font does not map in any way to the standard unit of type size measure called a "point," which is about 1/72 of an inch.) For the CGA, the value will always be equal to 8; for the MDA, it will be equal to 14. For the EGA and VGA it could be either, or for the VGA it could also be 16; an additional query must be made to figure out the point size for the more advanced adapters. Again, a call to BIOS VIDEO accomplishes this through the EGA BIOS extension service $11, Character Generator Functions. Subfunction $30, Return Information, provides the number of bytes per character (and hence the number of scan lines, because all characters in IBM text displays are 8 bits, or 1 byte, wide) in register CX.

```
{->>>>DeterminePoints<<<<-----------------------------------}
{                                                           }
{ Filename : FONTSIZE.SRC -- Last Modified 7/11/88          }
{                                                           }
{ This routine determines the character cell height for the }
{ font currently in use.  For the MDA and EGA this is hard- }
{ wired; for the EGA, VGA, and MCGA the value must be obtained }
{ by querying the ROM BIOS.                                 }
{                                                           }
{                                                           }
{                                                           }
{-----------------------------------------------------------}

FUNCTION DeterminePoints : Integer;

VAR
  Regs : Registers;

BEGIN
  CASE QueryAdapterType OF
    CGA       : DeterminePoints := 8;
    MDA       : DeterminePoints := 14;
    EGAMono,         { These adapters may be using any of   }
    EGAColor,        { several different font cell heights, }
    VGAMono,         { so we need to query the BIOS to find }
    VGAColor,        { out which is currently in use. }
    MCGAMono,
    MCGAColor : BEGIN
                  WITH Regs DO
                    BEGIN
                      AH := $11;  { EGA/VGA Information Call }
                      AL := $30;
                      BL := 0;
                    END;
                  Intr($10,Regs);
                  DeterminePoints := Regs.CX
                END
  END  { CASE }
END;
```

The main use of **DeterminePoints** is in turning the cursor on after it has been suppressed. The following procedure turns the cursor on, assuming the default text cursor of the two bottommost scan lines of the character cell. All font sizes are covered by specifying the cursor starting scan line as **Points-2** and the ending scan line as **Points-1**. The bottom scan line is not given by **Points** alone because **Points** gives the *number* of scan lines in the cell, and VIDEO requires the ordinal value of a line, counting from zero. For example, for the MDA font, there are 14 lines in the character cell, and the lines are numbered from 0 through 13.

```
{<<<< CursorOn >>>>}
{                                                                    }
{                                                                    }
{                                                                    }
{ HIGHLY specific to the IBM PC }

PROCEDURE CursorOn;

VAR
  Points : Byte;
  Regs   : Registers;

BEGIN
  Points := DeterminePoints;
  Mem[$40:$87] := Mem[$40:$87] OR $01;
  WITH Regs DO
    BEGIN AX := $0100; CH := Points-3; CL := Points-1; END;
  INTR(16,Regs);
END;
```

Note that this procedure **CursorOn** differs significantly from the procedure **CursorOn** that I described in earlier editions of *Complete Turbo Pascal,* which was written before either the EGA or VGA came into wide use.

Font Size and Screen Size

We've become so used to 25-line screens that we often forget that the EGA and VGA have the ability to display other, larger screen sizes. The EGA can display a 43-line screen, and the VGA can display a 43- or 50-line screen.

It's all done with fonts. As we've seen, the size in pixels of an IBM display adapter font may be 8, 14, or 16. The default font on an EGA is 14 pixels high. The default font on the VGA and MCGA is 16-pixels high. However, the EGA and VGA can both use an 8-pixel font, and when they do, they display 43 or 50 lines, respectively.

There is no separate "mode" for 43- or 50-line screens. You set the screen size simply by changing the fonts. This is done through yet another EGA/VGA/MCGA BIOS call.

The program FONT.PAS not only demonstrates how to select fonts on the EGA/VGA/MCGA, but provides several useful services as well. It identifies the installed display adapter. (On monochrome systems, which adapter is installed is not always obvious at a glance.) It identifies the current font size and passes that value to DOS ERRORLEVEL, so that batch files can take action based on the current font size (and hence screen) size. Finally, it allows you to enter a font size on the command line and change to that font size.

As with all good utility programs, the bulk of **Font**'s logic is user error trapping. **Font** determines the installed display adapter, and it does not allow a font change BIOS call to be attempted on an adapter that does not support a given font size. For example, trying to load the 16-pixel font on an EGA is not allowed.

The only "magic" in **Font** involves the suppression of BIOS cursor emulation. IBM's original EGA BIOS could not emulate the earlier cursors due to a rather blatant bug in the BIOS logic. For this reason, IBM did not push 43-line mode on the EGA. To take control of the cursor style you have to remove the BIOS's own grip on the cursor, and this is done by setting a bit in a location in low memory:

```
MEM[$40:$87] := MEM[$40:$87] OR $01;
```

The only caution here is that some inexpensive third-party EGA clone adapters have even buggier BIOSes that may react peculiarly to this kind of treatment. If your EGA goes berserk when you run **Font** for an 8-pixel font, you will not be able to use 43-line mode, which depends on the 8-pixel font. Get a better adapter; I have found both Paradise and Video 7 to be accurate copies of IBM's original EGA, right down to IBM's BIOS bugs.

Greater love than that, no clone hath.

```
{--------------------------------------------------------------}
{                              FONT                            }
{                                                              }
{      Display adapter text font query and change utility      }
{                                                              }
{                              by Jeff Duntemann               }
{                              Turbo Pascal V4.0               }
{                              Last update 7/1/88              }
{                                                              }
{                                                              }
{                                                              }
{--------------------------------------------------------------}

PROGRAM Font;

USES Crt,DOS;
```

```

TYPE
  AdapterType = (None,MDA,CGA,EGAMono,EGAColor,VGAMono,
                 VGAColor,MCGAMono,MCGAColor);
  FontSizes   = SET OF Byte;

CONST
  AdapterStrings : ARRAY[AdapterType] OF String =
                   ('None','MDA','CGA','EGAMono','EGAColor',
                    'VGAMono','VGAColor','MCGAMono','MCGAColor');


VAR
  InstalledAdapter : AdapterType;
  LegalSizes       : FontSizes;
  AdapterSizes     : FontSizes;
  ErrorPos         : Integer;
  ErrorSize        : String;
  NewFont          : Byte;
  FontCode         : Byte;
  OldAdapters      : SET OF AdapterType;
  Regs             : Registers;


{$I QUERYDSP.SRC}  { Contains function QueryAdapterType; see Section 18.4 }

{$I FONTSIZE.SRC}  { Contains function DeterminePoints; see Section 18.4 }


PROCEDURE ShowFontSizeError(BadSize : String);

BEGIN
  Writeln(BadSize,' is not a valid font size.');
  Writeln('Legal values are 8, 14, and 16,');
  Writeln('*if* your display adapter supports them.')
END;



BEGIN  { MAIN }
  LegalSizes := [8,14,16];  { IBM adapters only use these three sizes }
  OldAdapters := [CGA,MDA]; { The CGA and MDA cannot change fonts }

  IF ParamCount < 1 THEN
    BEGIN
      InstalledAdapter := QueryAdapterType;
      Writeln('>>FONT<<  V1.1 by Jeff Duntemann');
      Writeln('          From the book, COMPLETE TURBO PASCAL 5.0');
      Writeln('          ISBN 0-673-38355-5');
      Writeln;
      Writeln('The installed adapter is: ',
               AdapterStrings[InstalledAdapter]);
      Writeln('The current font size is: ',DeterminePoints);
      Writeln;
      Writeln
      ('To change the current font size, invoke FONT.EXE with the desired');
      Writeln
      ('font size as the only parameter, which must be one of 8, 14, or 16:');
```

```
        WRiteln; Writeln('   C>FONT 14'); WRITELN;
        Writeln('Remember that the font size of the CGA and MDA cannot change.');
        Writeln
  ('The EGA supports 8 and 14, while the VGA supports 8, 14, or 16.');
        Writeln('The MCGA supports the 16 pixel font size *only*.');
        Writeln
  ('FONT.EXE passes the current font size in ERRORLEVEL for use in batch files.');
        Halt(DeterminePoints)  { Make point size available in ERRORLEVEL }
        { THIS IS AN EXIT POINT FROM FONT.PAS!!! }
      END
    ELSE
      BEGIN
        Val(ParamStr(1),NewFont,ErrorPos);
        IF ErrorPos <> 0 THEN ShowFontSizeError(ParamStr(2))
        ELSE
          IF NOT (NewFont IN LegalSizes) THEN
            BEGIN
              Str(NewFont,ErrorSize);
              ShowFontSizeError(ErrorSize)
            END
          ELSE      { At this point entered font size is OK... }
            BEGIN   { ...but we must be sure the adapter supports it: }
              InstalledAdapter := QueryAdapterType;
              CASE InstalledAdapter OF
                CGA                 : AdapterSizes := [8];
                MDA                 : AdapterSizes := [14];
                EGAMono,EGAColor    : AdapterSizes := [8,14];
                VGAMono,VGAColor    : AdapterSizes := [8,14,16];
                MCGAMono,MCGAColor : AdapterSizes := [16];
              END;  { CASE }
              IF NOT (NewFont IN AdapterSizes) THEN
                BEGIN
                  Writeln('That font size does not exist');
                  Writeln('on your display adapter.')
                END
              ELSE      { Finally, do the font switch }
                BEGIN
                  ClrScr;
                  IF NOT (InstalledAdapter IN OldAdapters) THEN
                    BEGIN
                      CASE NewFont OF
                        8  : FontCode := $12;
                        14 : FontCode := $11;
                        16 : FontCode := $10;
                      END;  { CASE }
                      Regs.AH := $11;  { EGA/VGA character generator services }
                      Regs.AL := FontCode;  { Plug in the code for this size... }
                      Regs.BX := 0;
                      Intr($10,Regs);  { ...and make the BIOS call. }
                      { Suppress BIOS cursor emulation: }
                      MEM[$40:$87] := MEM[$40:$87] OR $01;
                      { Now reset the cursor to the appropriate lines: }
                      Regs.AX := $100;
                      Regs.BX := 0;
                      Regs.CL := 0;
                      Regs.CH := NewFont - 2;  { i.e., 6, 12, or 14 }
                      Intr($10,Regs);  { Make the BIOS call. }
                      HALT(DeterminePoints);
                  END
              END
          END
    END
END.
```

19 Console, Printer, and Disk I/O

A computer program is a universe unto itself. Within its bounds, data structures are created, changed, and deleted. Calculations are performed and the results used in still more calculations. Lots goes on in a computer program's universe during the process of getting a job done.

Out here, though, in *our* universe, is where the job comes from and where the finished work is needed. Somehow some of the critters living in a computer program have to cross the threshold between the program universe and the real universe. The pathways between the computer program and the outside world are collectively called "input/output" or, more tersely, "I/O." In Pascal, all I/O is handled under the umbrella category of files.

To most people, the word "file" conjures images of filing cabinets and various other places to store things away for future perusal and re-use. However, another connotation of the word "file" strikes closer to heart of the Pascal file: a long row of people or objects in a straight line, often in the process of moving from one place to another in an orderly fashion. Soldiers file from the parade ground to the mess hall, and so on.

A Pascal file is a stream of data, either coming or going. Not all "files" are data stored on disk or tape. Your keyboard is a file: a stream of characters triggered by your fingers and sent inward to the waiting program. Your CRT screen is a file: a flat field painted with visible symbols by a stream of characters sent from the program inside the computer. Your printer is a file: a device that accepts a stream of characters from the program and places them somehow onto a piece of paper.

The last example contains an important word: *device.* Your keyboard is a device, as is your CRT screen, as is your printer, and as is your modem. All are files that have one endpoint in the computer program and another endpoint in a physical gadget that absorbs or emits data (sometimes a file may do both). This kind of file we call a "device file."

The other kind of file is the more familiar kind: a collection of data recorded as little magnetic disturbances on a clean piece of magnetic plastic. These are *disk files.* Files in Turbo Pascal are of one type or another, either device files or disk files.

19.1: LOGICAL AND PHYSICAL FILES

ISO Standard Pascal does not make a distinction between device and disk files. In ISO Pascal, a file is a file—a stream of data either coming or going. One end of the line is inside the Pascal program, and Pascal simply doesn't care what sort of thing is on the other end of the line.

Turbo Pascal has to modify this notion a little—for random files, as an example—but the "everything Out There is a file" bias remains. By not getting involved in the deep-down details of how a character is printed or sent to the screen, Pascal can present a uniform front to many different operating systems. This makes for less work in translating a Pascal program from one operating system environment to another.

This separation between *what* a file does and *how* it does it is sharply drawn in Pascal, between the concepts of *logical* and *physical* files.

A logical file is a Pascal abstraction, beginning with the declaration of a file variable in the variable declaration part of a program:

```
StatData : FILE OF Integer;
```

This tells us that **StatData** is a file and contains integers, nothing more. It says nothing about whether or not the file exists on a disk, which disk, or how much data exist in the file.

That information falls under the realm of physical files. A physical file is an actual device (for device files) or an actual collection of data on some sort of storage medium. A physical disk file has a file name, a size, a record length, an access mode, (read only, read/write, hidden, system, etc.) perhaps a timestamp for last access and an interleave factor, and other things as well. It is *not* an abstraction, and by not being an abstraction a physical file must pay attention to all those gritty little details that make storing and retrieving data on disk possible.

All these little details are very much hardware- and operating system-dependent. If Pascal had to take care of such details, it would be nearly an entirely separate language for each computer/operating system combo it ran on, as is very nearly the case these days with BASIC and it has always been the case with FORTH.

Thus, Pascal provides logical files, and the operating system provides physical files. When a logical file becomes associated with (or *assigned to* as is often said) a physical file, a path exists between the program's inner universe and the outside world. That file is then said to be *open.*

19.2: DECLARING, ASSIGNING, AND OPENING FILES

Declaring a file gives the file a name and a data type:

```
FileVariableName : FILE OF <type>;
```

<type> may be any valid data type except a file type. A file of files makes no logical sense and is illegal. This <type> can be a standard type like **Integer** or **Boolean**, or it can be a type that you have defined, like records, sets, enumerated types or arrays. A file can be a file of only one type, however; you cannot declare a file of **Integer** and then write records or sets to it.

Declaring a file also creates a file buffer in your data area. This file buffer is a variable of the type the file is declared to be. It is a *window* into the file; the actual open end of the data pipe between the program and the outside world.

Unlike ISO Standard Pascal and most other commercial implementations of Pascal, Turbo Pascal does *not* allow direct access to the file window via **Get**, **Put**, or the caret notation:

```
VAR
  NumFile : FILE OF Integer;

NumFile^ := I;    { Illegal in Turbo; OK in Standard Pascal }
Put(NumFile);
```

Connecting Logical to Physical with Assign

Before you can open a file, there must first be a connection between a logical file that is declared in the program and a physical file that exists outside the program. Creating this connection is the job of the built-in procedure **Assign**.

Assign takes two parameters: a file variable that has been declared in the program and a string variable or literal that names a physical file:

```
PROCEDURE Assign(AFile : <filetype>; FileName : String);
```

Assign creates the File Information Block (usually called a FIB), which each file must have to be used by the program. Once associated with a physical file, a logical file may be opened for write access or read access. This is done with **Reset** and **Rewrite**.

Opening Files with Reset and Rewrite

The **Assign** procedure is *not* part of ISO Standard Pascal. Most commercial Pascal compilers use it as the means of connecting logical to physical files. However, **Assign** does only half the job of opening a file; its job is limited to linking a logical file with a physical file. Getting a logical file ready to be written to or read from involves two built-in procedures: **Reset** and **Rewrite**. These procedures are part of ISO Standard Pascal.

An open file has in its FIB what we call a *file pointer*. This is not a pointer variable as we know it, but rather a logical marker indicating which element in the file will be read or written to next.

Reset opens a file for reading. Some device files allow both read access and write access at the same time; these files are opened with **Reset** also. When a file is to be opened for random file I/O using **Seek** (see Section 19.9), **Reset**, again, is used.

Reset does not disturb the previous contents of a disk file. The file pointer is positioned at the first data item in the file. If the file is empty when **Reset** is executed, the **EOF** function returns **True** for that file.

Some examples:

```
Assign(MyFile,'A:ADDRESS.TXT');
Reset(MyFile);

Assign(ConsoleInput,'CON');
Reset(ConsoleInput);
```

Reset can also be used on a file that is already open. For an open disk file, this will reposition the file pointer to the first record in the file. Performing a **Reset** on an open device file does nothing.

Rewrite opens files that are *only* to be written to, and not read from while they are currently open. When **Rewrite** is executed on a disk file, all previous contents of the disk file are overwritten and lost. Note that you may use either **Reset** or **Rewrite** on device files; the action on the device file is identical for both.

```
Assign(MyFile,'B:GRADES.DAT');    { Old grades are lost! }
Rewrite(MyFile);

Assign(BitBucket,'NUL');        { Anything written to BitBucket }
Rewrite(BitBucket);             { will quietly vanish away...   }
```

Device Files and DOS Device Names

All files have names. You're certainly familiar with the list of file names that comes up when you use the DIR facility of your operating system. DIR displays a directory of disk file names on a given disk drive. Device files have names as well, if not such easily displayable ones.

Each operating system has its own means of getting information to and from devices. Most operating systems have a set of standard devices that are supported by function calls to the operating system BIOS (Basic Input/Output System.) Turbo Pascal recognizes names for all devices supported by the host operating system. These names are used by the Turbo Pascal file handling machinery just as names for disk files are used.

Table 19.1 summarizes the DOS device names that are useable in Turbo Pascal.

Table 19.1
DOS Device Names in Turbo Pascal

Name	*Device definition*
CON	Buffered system console. Echoes input characters CR as CR/LF; BS as BS/SP/BS. As OUTPUT, echoes CR as CR/LF; LF alone is ignored. HT (horiz. tab) is expanded to every 8 columns. CAN (CRL-X) erases input line and returns cursor to its starting column.
LPT1 LPT2 LPT3	These are output-only devices generally attached to PC parallel ports. PRN is a synonym for LPT1. The Printer unit assigns LPT1 to a text file variable named LST.
COM1 COM2	These refer to the PC's standard serial ports. There is no status call; if you read one of these devices when no character is ready it will wait for a character, effectively hanging the system.
NUL	"Bit bucket" device. Absorbs all output without effect. Generates EOF immediately on any read.

Be careful with COM1 and COM2. The fact that they are input/output devices implies that they could be used to write a "dumb terminal" program to read characters from keyboard and send them to the modem and read characters from the modem and send them to the screen. Unfortunately, there is no way to test COM1 or COM2 to see if a character is ready to be read before you actually go in and read one. And if a character is *not* ready, the **Read** statement will wait until one is ready, effectively "hanging the system" until something comes in on the modem line. In practice, you need to go to the BIOS or to the hardware to do effective input from COM1 or COM2, because you need to test for the *presence* of a character before you try to *read* a character.

These device names are *not* files and cannot be used as file identifiers. To use them, you must assign them to a Pascal file variable identifier, and then either reset or rewrite them as appropriate.

Input and Output

All Pascal programs are capable of accepting input from the keyboard and writing output to the CRT screen. Two standard device files are always open for this purpose: **Input** and **Output**.

In some Pascals you must explicitly name **Input** and **Output** when using them with **Read**, **Write**, **Readln**, and **Writeln**:

```
Writeln(Output,'Files do it sequentially...');
```

In Turbo Pascal, **Input** and **Output** are the default files for use with **Read**, **Write**, **Readln**, and **Writeln**. If you omit the name of a file in one of those statements, the compiler assumes **Input** for **Read** and **Readln**, and **Output** for **Write** and **Writeln**. You need only write:

```
Writeln('Files do it sequentially...');
```

and Turbo Pascal will know to send the text line to your CRT screen.

Similarly, to accept input from the **Input** logical file, which is always connected to your keyboard and always open, you need only write:

```
Readln(AString);
```

and the program will wait while you type text at the keyboard, going on only when it detects a carriage return or when you have filled the string variable out to its maximum physical length. The text you typed will be immediately available in the variable **AString**. There is no need to include the identifier **Input** in the **Readln** statement, although you may if you wish.

Input and **Output** are considered text files, which means that only printable ASCII characters and a few control characters may be read from them or written to them.

However, **Read**, **Readln**, **Write**, and **Writeln** contain limited abilities to convert binary data types like **Integer** to printable representations. So it is in fact legal to **Write** an integer to **Output** because **Write** converts the integer to printable characters before **Output** ever sees it. For a more complete discussion, see Section 19.7, "Using **Read** and **Write** with text files."

Under Turbo Pascal 4.0 and later versions, **Input** and **Output** are assigned to DOS standard input and DOS standard output, respectively. This means that characters written to **Output** may be redirected to a DOS file by use of the DOS redirection feature. Also, characters read from **Input** may be redirected to come from a DOS disk file instead of from the console keyboard.

An important thing to remember about **Input** and **Output** is that when you use the **CRT** unit (see Chapter 18), they are no longer assigned to DOS standard input and standard output but are assigned directly to Turbo Pascal's low-level screen and keyboard support. This means that DOS redirection will not work once you **USE** the **CRT** unit.

Binary Files Versus Text Files

All disk files eventually can be seen as collections of bytes grouped somehow on a diskette or hard disk. Pascal makes a distinction between files that can contain absolutely any binary pattern in a stored byte and those files that are allowed to contain only printable characters and certain (very few) control characters. Files that may contain any byte pattern at all are called "binary" or "nontext" files. Files that are limited to printable characters are called text files.

Text files are allowed to contain printable ASCII characters plus whitespace characters. Whitespace characters include carriage return ($0D), linefeed ($0A), formfeed ($0C), horizontal tab ($09), and backspace ($08). Although not technically a whitespace character, the bell character ($07) also is permitted in text files.

Text files may be "typed" (via the operating system TYPE command) directly from disk to the screen without sending the display controller into suicide fits.

An important consequence of allowing only printable characters is that text files created under PC DOS may contain an end-of-file (EOF) marker character. Binary files may *not* contain an EOF marker because binary files are allowed to have any 8-bit character pattern as valid data. Therefore the operating system (and thus Turbo Pascal) could not tell an EOF marker character from just more legal binary data.

Text files under the DOS operating system have traditionally used CTRL-Z (hex 1A) as the EOF marker. This is a holdover from DOS's origins in CP/M-80, which did *not* keep track of the size of files down to the byte, as DOS does. CP/M-80 only knew how many 128-byte blocks were in a file; to mark the true end of data within the last data block, an application had to place a CTRL-Z character after the last true byte of data.

Turbo Pascal writes a CTRL-Z after the last data byte in a text file. Furthermore, when reading a text file, it reports EOF at *either* the true EOF reported by DOS *or* at the first CTRL-Z character found. This could be important if you allow a CTRL-Z character to be written somewhere in the middle of a text file, perhaps by way of the **BlockWrite** procedure. If you later read that file through **Read** or **Readln**, you will be able to read *only* up to the CTRL-Z, at which point EOF will become true.

Text files in Turbo Pascal are declared this way:

```
VAR
  MyFile    : FILE OF Text;     { Both forms are legal }
  YourFile : Text;
```

Any file that is not declared to be a **FILE OF Text** or simply **Text** is considered a binary file. The difference between the two becomes critical when you need to detect where data ends in the file. We will examine this problem in detail in connection with the EOF function in Section 19.4.

19.3: READING FROM AND WRITING TO FILES

It is in the realm of file I/O that Turbo Pascal deviates most from ISO Standard Pascal. Standard Pascal has a pair of file procedures called **Get** and **Put**. These work with a data structure called the "file window," which is a slice of the file to be read from or written to. When you **Get** from a file, the next item in the file is placed in the file window, from which it may be accessed at leisure. Similarly, when you wish to write an item to a file, you place it in the file window and then **Put** the file window out to the disk.

Turbo Pascal does *not* implement **Get** and **Put**, nor does it make the file window directly available to the programmer. If you are porting Pascal code from another compiler, all **Gets**, **Puts**, and file window references will have to be edited to use **Read** and **Write** instead.

Using Read and Write with Non-Text Files

We won't say much more about **Get** and **Put**. They do nothing that can't be done with **Read** and **Write**. That, their awkwardness, and the need to keep the size of the compiler down were the reasons Turbo Pascal does not implement **Get**, **Put**, and the file window concept of file I/O.

The following discussion deals with "typed" or binary files that are not files of **Text**. Text files are a special case and will be covered in the next section.

Once a file has been opened with **Reset**, data may be read from the file with the **Read** procedure. **Read** takes at least two parameters: a file variable and one or more variables with the same type as the file:

```
VAR
  StatRec,Rec1,Rec2,Rec3  : NameRec;
  MyFile : FILE OF NameRec;

Assign(MyFile,'B:NAMES.DAT');
Reset(MyFile);

Read(MyFile,StatRec);
Read(MyFile,Rec1,Rec2,Rec3);
```

Here, **NameRec** is a record type defined earlier in the program. In this example, the **Read** statement reads the *first* record from **MyFile** into the record variable **StatRec**. There is no mention of the file window variable, although the file window is involved beneath the surface in the code that handles the **Read** function. The second **Read** statement reads the next three elements of **MyFile** into **Rec1**, **Rec2**, and **Rec3**, all at one time. This is a convenient shorthand; there is no difference in doing this than in doing three distinct **Read** statements.

Write works just the same way with the same parameters; the only difference is the direction in which the data are flowing:

```
Assign(MyFile,'B:NAMES.DAT');
Rewrite(MyFile);

Write(MyFile,StatRec);
Write(MyFile,Rec1,Rec2,Rec3);
```

Read and **Write** work *sequentially;* each time **Read** or **Write** is used, the file pointer is bumped to the next element down the file. The process never works backwards; you cannot begin at the end of the file and work your way back. Again, for that sort of thing you need random file I/O (see Section 19.9).

Using Read and Write with Text Files

A text file is a stream of printable characters between the program and a device file or disk file. The **Read** and **Write** statements are used to transfer a data item (or list of data items) to or from the text file character stream. The data items do not have to be of the same type. We've been doing things like this informally all through this book:

```
VAR
  Unit  : Char:
  Count : Integer;

Write('The number of files on disk ',Unit,' is ',Count);
```

What is happening here is that a list of four data items is being sent to device file **Output**. Two are string literals, one is a character, and one is an integer.

Text files, as we saw in the previous section, may contain only printable ASCII characters and certain control characters. Integer variables are not ASCII characters; they are 2-byte binary numbers that are not necessarily printable to the screen.

Yet when you write:

```
VAR
  I : Integer;

I := 42;
Write(I);
```

the two ASCII characters 4 and 2 appear on your screen. They are an ASCII representation of a 2-byte binary integer that could as well have been written in base 2 as 00000000 00101010. **Read** and **Write** contain the machinery for converting between numeric variables (which are stored in binary form) and printable ASCII numerals.

Write also has the ability to take Boolean values (which are actually binary numbers 0 and 1) and convert them to the words "TRUE" and "FALSE" before passing them to a text file. **Read**, however, does *not* convert the ASCII strings "TRUE" or "FALSE" to Boolean values!

Read and **Write** when used with text files can accept data types **Integer**, **Char**, **Byte**, **Word**, **LongInt**, **ShortInt** and subranges of those types; type **String** and derived string types; plus type **Real**, **Single**, **Double**, **Extended**, and **Comp**. **Write** will also accept Boolean values; remember that **Read** will not. Enumerated types, pointers, sets, arrays, and record types will generate errors during compilation. Attempting to **Read** a Boolean value from a text file will trigger this error:

```
Error 64: Cannot Read or Write variables of this type
```

Although **Read** will accept a string variable, it's better practice to use **Readln** for reading strings (see below on **Readln**.) The problem is that strings on a text file character stream carry no information about how long they are. If you **Read** from a text file stream into a string variable, the string variable will accept characters from the file until it is physically filled. If the file ends before the string is completely filled, the system may hang or fill the remainder of the string with garbage.

Readln and Writeln

Pascal includes a pair of I/O procedures that work only on text files: **Readln** and **Writeln**. They introduce a whole new concept: dividing a Pascal text file into "lines."

Ordinary text on a printed page is a series of lines. A Pascal text file is a one-dimensional stream of characters; it has none of the two-dimensional quality of a printed page. To model text as we see it in the real world, Pascal defines an end-of-line character (EOL) that is inserted into the character stream of a text file after each line of printable characters. The definition of the EOL character may vary from system to system. For most microcomputers, EOL is not one character but two: the sequence carriage return/line feed ($0D/$0A). This is a holdover from teletype's heyday when it took one control character to return the typehead to the left margin and yet another to index the paper up one line.

The **Writeln** procedure is exactly like **Write**, except that it follows the last item sent to the text file character stream with the EOL character. For our discussion EOL will always be the pair CR/LF.

```
PROGRAM WriteInt;

VAR
  IntText : Text;
  I,J     : Integer;

BEGIN
  Assign(IntText,'B:INTEGERS.DAT');
  Rewrite(IntText);
  FOR I := 1 TO 25 DO Writeln(IntText,I);
  Close(IntText);
END.
```

This program writes the ASCII equivalent of the numbers from 1-25 to a text file. Each numeral is followed by a CR/LF pair. A hexdump of file INTEGERS.DAT will allow you to inspect the file character stream:

```
0000  31 0D 0A 32 0D 0A 33 0D   0A 34 0D 0A 35 0D 0A 36
0010  0D 0A 37 0D 0A 38 0D 0A   39 0D 0A 31 30 0D 0A 31
0020  31 0D 0A 31 32 0D 0A 31   33 0D 0A 31 34 0D 0A 31
0030  35 0D 0A 31 36 0D 0A 31   37 0D 0A 31 38 0D 0A 31
0040  39 0D 0A 32 30 0D 0A 32   31 0D 0A 32 32 0D 0A 32
0050  33 0D 0A 32 34 0D 0A 32   35 0D 0A 1A 1A 1A 1A 1A
0060  1A 1A 1A 1A 1A 1A 1A 1A   1A 1A 1A 1A 1A 1A 1A 1A
0070  1A 1A 1A 1A 1A 1A 1A 1A   1A 1A 1A 1A 1A 1A 1A 1A
```

If you know your ASCII well (or have a table handy), you can see the structure of the character stream in this file: ASCII numerals separated by 0D/0A pairs. Turbo Pascal fills out the 128-byte block after the end of data with CTRL-Z characters. In text files, the first CTRL-Z signals end of file; more on that shortly.

In a sense, **Writeln** writes only strings (minus length bytes) to its files. If you hand it a variable that is not a string, **Writeln** will (if the conversion is possible) convert it to a string before writing it to its file.

Readln reads one line from a text file. A line, again, is a series of characters up to the next EOL marker. If **Readln** is reading into a string variable, the number of characters read becomes the logical length of the string. So, unlike **Read** and **Write**, **Readln** and **Writeln** can in fact maintain information on string length in a file, because all strings written by **Writeln** are bounded by EOL markers.

Formatting with Write Parameters

Write and **Writeln** allow certain formatting options when writing data to text files, including the screen (**Output**) and system printer. These options as given to the **Write** and **Writeln** statements are called *write parameters.*

The simplest write parameter applies to any variable that may be written to a text file and specifies the width of the field in which the variable is written:

```
<any Writeable variable> : <field width>
```

When written, any data in the variable will be right justified within a field of spaces, <field width>, wide. For example:

```
CONST
  Bar = '|';

VAR
  I    : Integer;
  R    : Real;
  CH   : Char;
  OK   : Boolean;
  Txt  : String;

I := 727;
CH := 'Z';
R := 2577543.67;
OK := False;
Txt := 'Grimble';

Writeln(Bar,I:5,Bar);
Writeln(Bar,CH:2,Bar);
Writeln(Bar,R:12,Bar);
Writeln(Bar,OK:7,Bar);
Writeln(Bar,TXT:10,Bar);
Writeln(Bar,-R,Bar);
Writeln(Bar,R,Bar);
```

When run, this code snippet produces this output:

```
|  727|
| Z|
| 2.577544E+06|
|  FALSE|
|   Grimble|
|-2.57754E+06|
| 2.57754E+06|
```

Note that the real numbers are always expressed in exponential (also called "scientific") notation, that is, as powers of ten, even though originally expressed with a decimal point and no exponent. To express a real number without the exponent, you must include a second write parameter for the width of the decimal part of the field:

```
<real value> : <field width> : <decimal width>
```

The value <decimal width> indicates how many decimal places are to be displayed. For example:

```
R := 7.775;
S := 0.123456789;
T := 7765;

Writeln(Bar,R:10:3,Bar);
Writeln(Bar,R:10:1,Bar);
Writeln(Bar,R:5:2,Bar);
Writeln(Bar,S:6:6,Bar);
Writeln(Bar,S:12:6,Bar);
Writeln(Bar,S:12:12,Bar);
Writeln(Bar,S:5,Bar);
Writeln(Bar,T:5:2,Bar);
Writeln(Bar,T:5,Bar);
Writeln(Bar,T:6:6,Bar);
```

This code produces the following output:

```
|     7.775|
|       7.8|
| 7.77|
|0.123457|
|    0.123457|
| 1.2345678E-01|
| 1.E-01|
|7765.00|
| 8.E+03|
| 7.7650003E+03|
```

A reminder here: You cannot begin a fractional real number (such as .123456789, above) with a decimal point. You *must* begin the number with a 0 as shown or suffer this error during compilation:

```
Error 42: Error in expression
```

Writing to the System Printer

Your printer is a physical device that may be accessed as a text file. You have the option of assigning one of the DOS device names LPT1, LPT2, LPT3, COM1, or COM2 to a text file of your choosing and **Reset**ing that text file, or using the standard unit **Printer**. **Printer** assigns DOS device name LPT1 to a text file named **LST** and opens **LST** before the main program begins execution. Thus **LST** will be immediately available to your code without any action from you other than to use the **Printer** unit. Either way you set up a printer text file, writing any text to that text file will print the text on your printer.

```
PROGRAM PrinTest;

USES Printer;

VAR
  PrintDevice : Text;

BEGIN
  Assign(PrintDevice,'LPT1');        { Open a printer file }
  Rewrite(PrintDevice);

  Writeln(PrintDevice,'This text will now appear on the printer.');
  Writeln(LST,'This will too, and it''s easier!');
END.
```

Using **Writeln** to the printer will end each line with a carriage return/linefeed pair that will bring the printhead to the left margin of the next line. Using **Write** will leave the printhead where it is after printing the text:

```
Write(LST,'Active drive units now include: ');
FOR Drive := 'A' to 'P' DO
  IF CheckDrive(Drive) THEN Write(LST,Drive,' ');
Writeln(LST,' ');
```

This will print:

```
Active drive units now include: A B D M N
```

Even if you open a printer file yourself (rather than use the **Printer** unit's file **LST**) you need not close the file.

19.4: IOResult AND EOF

IOResult

Working with creatures like disk files that lie outside the borders of the program itself is risky business. You can build machinery into your program to make sure that the program never attempts to divide by 0 or to index past the bounds of an array. But how do you make sure that, when you want to open a disk file, the file is on the disk and the disk is in the proper disk drive?

The program does not have absolute control over disk files in the way it has absolute control over numbers, arrays, and other variables. The only way to be sure a file is on the disk is to go out and try to read it. If the file isn't out there, there must be some way to recover gracefully.

The runtime code that Turbo Pascal adds to every program it compiles guards against runtime errors such as an attempt to open (for reading) a file that does not exist. Such an attempt will generate an I/O error 01: File does not exist. Your program will terminate, and, if you are running from within the Turbo Pascal Environment, Turbo will begin searching for the location of the **Reset** statement that tried to open the file.

Obviously, if there is a legal possibility of a file not existing on the disk, you cannot allow your program simply to crash. Better to determine that an error has happened *without* crashing, so that the program could do something about it, like create a new file or look somewhere else for the old one. Turbo Pascal provides the **IOResult** function to let the program know how successful it has been in striking a path to the outside world. It is predeclared by Turbo Pascal this way:

```
FUNCTION IOResult : Integer
```

After each I/O statement is executed, a value is given to the **IOResult** function. This value can then be tested to determine whether the I/O statement completed successfully. A 0 value indicates that the I/O operation went normally; anything else constitutes an error code.

However, the runtime code's error traps will crash your program when an error is encountered whether or not you use **IOResult**. To keep the program running in spite of the error, you must disable the error trap with the compiler directive **$I**. This is done by surrounding the I/O statement with an **{$I−}** directive (turn traps off) and an **{$I+}** directive (turn traps on again.)

For example:

```
Assign(MyFile,'B:BOWLING.DAT');
{$I-} Reset(MyFile); {$I+}  { Suspend error traps during Reset }
IF IOResult <> 0 THEN
  BEGIN
    Beep;
    Writeln('>>The bowling scores file cannot be opened.');
    Writeln('  Make sure the scores disk is in the B: drive');
    Writeln('  and press (CR) again:')
  END;
```

This code snippet checks **IOResult** immediately after executing a **Reset** to open a file for read. If **IOResult** returns a nonzero value, the program displays a message to the operator.

IOResult is cleared to 0 and refilled with a new value at the beginning of *all* I/O primitives. This includes **Readln** and **Writeln** when used (as we have been using them all along) to talk to the keyboard and the screen via device files **Input** and **Output**. This means you *cannot* directly write the value returned by **IOResult** to the screen or to a file!

```
{$I-} Reset(MyFile); {$I+}
Writeln('The result of that Reset is ',IOResult);  { No! }
```

No matter what happens when **Reset(MyFile)** is executed, the above snippet will *always* display:

```
The result of that Reset is 0
```

Given that restriction, it is probably good practice to assign the value of **IOResult** to an integer variable immediately after an I/O statement and work with the integer variable rather than the function itself.

Another example of **IOResult** in use lies in the **Averager** program in the next section. **Averager** opens and reads a binary file of integers until the end of the file and then averages the integers it has read. If it cannot open the file of integers it recovers and issues an error message with the help of **IOResult**.

EOF

Knowing where a disk file ends is critical. A built-in function called **EOF** (End Of File) provides this service to your programs. **EOF** is predeclared this way:

```
FUNCTION EOF(FileVar : <any file type>) : Boolean
```

FileVar can be any legal file type. The **EOF** function returns **True** as soon as the last item in the file has been read. At that point, the file pointer points just past the end of data in the file, and *no further reads should be attempted.* A runtime error will occur if you try to read beyond the end of a file. This applies to both text files and binary files.

Turbo Pascal's runtime code determines **EOF** by using information stored by DOS for each disk file. For text files, it keeps a count of characters read from the file and compares that with the size of the file at each read; when bytes read equals or exceeds the number of bytes DOS says are in the file, **EOF** returns **True**. For binary files, Turbo Pascal knows the size of each record read (since every file is a **FILE OF <something>**, where **<something>** is a declared data type with a fixed size) and compares a similar count of records read against the DOS file size figure.

Text file EOF is also triggered by the presence of a CTRL-Z character ($1A) in the file. For CP/M-80, this was the *only* way to determine true EOF, since CP/M only kept count of the 128-byte blocks in a file, not the actual number of bytes. The CTRL-Z was the only way to tell where within the last block of data the last significant character actually fell.

DOS knows exactly how many bytes are in every file at all times, so with DOS text files, Turbo Pascal reports EOF either when DOS indicated the last character had been read, or when a CTRL-Z was encountered in the text stream from the file.

The following program assumes a binary file of integers. (I provide such a file on the listings diskette for this book, or you can generate your own.) It reads all the integers in the file, testing **EOF** at each read. As it reads each integer, it keeps a running total and a running count and then produces an average value for all the integers in the file. Note the use of the **$I** compiler directive when opening the file:

```
{--------------------------------------------------------------}
{                           Averager                           }
{                                                              }
{            Binary file I/O demonstration program             }
{                                                              }
{                             by Jeff Duntemann                }
{                             Turbo Pascal V5.0                }
{                             Last update 7/24/88              }
{                                                              }
{                                                              }
{                                                              }
{--------------------------------------------------------------}

PROGRAM Averager;

VAR
  IntFile       : FILE OF Integer;
  I,J,Count     : Integer;
  Average,Total : Real;

BEGIN
  Assign(IntFile,'INTEGERS.BIN');
  {$I-} Reset(IntFile); {$I+}
  I := IOResult;
  IF I <> 0 THEN
```

```
      BEGIN
        Writeln('>>File INTEGERS.BIN is missing or damaged.');
        Writeln('  Please investigate and run the program again.')
      END
    ELSE
      BEGIN
        Count := 0; Total := 0.0;
        WHILE NOT EOF(IntFile) DO
          BEGIN
            Read(IntFile,J);
            IF NOT EOF(IntFile) THEN
              BEGIN
                Count := Count + 1;
                Total := Total + J
              END;
          END;
        Close(IntFile);
        AVERAGE := Total / Count;
        Writeln;
        Writeln('>>There are ',Count,' integers in INTEGERS.BIN.');
        Writeln('  Their average value is ',Average:10:6,'.');
      END
  END.
```

19.5: FileSize AND FilePos

Turbo Pascal supplies two built-in functions for determining the size of a binary disk file and the position of the file pointer within a disk file. Note that **FileSize** and **FilePos** may *not* be used with text files!

FileSize is predeclared this way:

```
FUNCTION FileSize(FileVar : <binary filetype>) : LongInt
```

The function **FileSize** returns a long integer count of the number of items stored in the file. **FileVar** is an open binary file, *not* a text file. An empty file will return a 0 value.

Function **FilePos** is predeclared this way:

```
FUNCTION FilePos(FileVar : <binary filetype>) : LongInt
```

A binary file's file pointer is a counter that indicates the next item to be read from the file. The **FilePos** function returns the current value of a file's file pointer. **FileVar** is an open binary file, *not* a text file. The first item in a file is item number 0. When a file is first opened, its file pointer is set to 0. Each **Read** operation will increment the file pointer by one. The **Seek** procedure (see Section 19.9) will put the file pointer to a particular value to enable random access to a binary file.

19.6: MISCELLANEOUS FILE ROUTINES

The last group of Turbo Pascal file routines we'll discuss are all extensions to the ISO Standard Pascal definition. Most Pascals have routines like these, but sadly, their invocation syntax and parameters vary widely. Any time you use file I/O, you can be almost certain that your programs will be nonportable. (There is no way even to close a file in ISO Pascal!) This is yet another reason why I have not stressed adherence to the ISO Standard in this book. If file I/O cannot be made portable, you might as well hang it up.

Flush

Every file has a buffer in memory, and when you write to a file, the data you've written actually go to the buffer rather than directly to the physical disk file. Periodically, based on decisions it makes on its own, the Turbo Pascal runtime code will flush the buffer to disk, actually transferring the data to the physical disk file. You can force such a flush to disk with the **Flush** procedure. It is predeclared this way:

```
PROCEDURE Flush(<filevar>);
```

<filevar> is any file variable opened for output. **Flush** has no effect on a file opened for input. **IOResult** will return a 0 value if the file was flushed successfully. *Do not use **Flush** on a closed file!*

Close

Closing a file that you have opened ensures that all data written to the file is physically transferred from the file buffer to disk. In Turbo Pascal the **Close** procedure does this job:

```
PROCEDURE Close(<filevar>);
```

As with **Flush**, **<filevar>** is any opened file. It is all right to close a file that has been closed already.

Closing files that had not been changed while open used to be optional. With Turbo Pascal 3.0 and later versions, this is not the case. You *must* close all opened disk files to keep the operating system happy. Turbo Pascal 3.0, 4.0, and 5.0 for PC/MS DOS use file handles, of which there are only sixteen available. File handles, once allocated by opening a file, will not be freed for further use until the file is closed. If you neglect to close a few temporary files, you may soon run out of file handles.

And of course, if you exit a program before closing a file that has been written to, the runtime code makes no guarantee that all records written to the file buffer will actually make it out to the physical disk file. As with most file-related routines, **IOResult** will be set to 0 if the file was closed successfully. Otherwise, **IOResult** will contain a nonzero DOS error code.

Erase

Deleting a disk file from within a program is done with the **Erase** procedure:

```
PROCEDURE Erase(<filevar>);
```

<filevar> is any file variable that has been assigned to a physical file. In other words, if you try to **Erase** a file variable to which no physical file has yet been assigned, the runtime code has no way of knowing which file you want to delete:

```
VAR
  NumFile : FILE OF Integer;

Assign(NumFile,'VALUES.BIN');
Erase(NumFile);
```

Do not attempt to **Erase** an open file. Close it first, or the results could be unpredictable. **IOResult** will return a 0 if the file was successfully deleted.

Rename

Turbo Pascal gives you the ability to change the name of a disk file from within a program with the **Rename** procedure:

```
PROCEDURE Rename(<filevar>; NewName : String80);
```

<filevar> is any file that has been assigned to a physical file with **Assign**. As with **Erase**, trying to rename a file without connecting it with a physical file is meaningless.

Use **Rename** with some caution. It is possible to rename a file to a name that another disk file already uses. No error will result, but you will have two files with the same name, and getting the *right* one when you need one of the two files will be a problem indeed.

As with **Erase**, renaming an open file is a no-no. **IOResult** will return a 0 if the file was successfully renamed.

Append

Append provides a quick way to move the file pointer to the end of a text file, without explicitly having to read the file, and to throw away the characters read up to EOF.

Append is used instead of **Rewrite**:

```
PROCEDURE Append(<filevar>);
```

<filevar> must first be assigned to some physical file before **Append** can be used. The contents of the file are not destroyed, as they are with **Rewrite**. The file is opened for output, however, at EOF. Adding text to the file with **Write** or **Writeln** will position the new text at and following EOF.

IOResult will return a 0 if the operation was successful. Keep in mind that **Append** may be used *only* with text files. Attempting to use **Append** with a nontext file type will trigger

```
Error 63: Invalid file type
```

Truncate

Truncate is conceptually similar to **Append**. Both prepare a file for the adding of additional data via **Write** or (for text files) **Writeln**. Unlike **Append**, **Truncate** may be used with any type of file: text, binary, or untyped. **Truncate** is predeclared this way:

```
PROCEDURE Truncate(<filevar>);
```

When executed, **Truncate** chops a file off at the current position of the file pointer. In other words, if you have read part way down a file and execute **Truncate**, the remainder of the file will be thrown away. The file is then ready for output, even if you opened the file with **Reset**.

Standard Pascal does not allow writing to a file that contains data. When a file is opened for output in Standard Pascal, all previous contents of the file are destroyed. This defies the logic of the real world, in which files are built by an ongoing process of reading, writing, updating, and deleting. **Append** and **Truncate** make real-world use of data files a great deal more convenient than in Standard Pascal.

19.7: USING TEXT FILES

Given its variable-length string type, its built-in string functions and procedures, and Standard Pascal's **Readln** and **Writeln** procedures, Turbo Pascal is a natural choice for working with text files. In this section we'll show you a real-life example of a useful program for manipulating text files.

Filter Programs

There is a whole class of programs that read a file in chunks, perform some manipulation on the chunks, and then write the transformed chunks back out to another file. This type of program is called a "filter" program because it filters a file through some sort of processing. The data change according to a set of rules as they pass through the processing part of the program.

A good example would be a program to force all lowercase characters in a file to upper case. Turbo Pascal is not sensitive to character case, but some programs and language processors (particularly COBOL and APL) do not interpret lowercase characters correctly. To pass a text file between Pascal and COBOL, all lowercase characters in the file must be set to uppercase.

A filter program to accomplish this task would work this way:

```
Open the input file for read and create a new output file.

While not end-of-file keep doing this:

   Read a line from the input text file.

   Force all lowercase characters in the line to uppercase.
   Write the line out to the output text file.

Close both files.
```

This basic structure is the same for all text file filter programs, except for the processing that is actually done line by line. You could just as easily force all uppercase characters in the line to lower case, count the words in the line, remove all BEL characters (CTRL-G) from the line, expand HT (tab; CTRL-I) characters to eight space characters, and so on.

You could, in fact, combine two or more processes into one filter program; say, force lowercase letters to uppercase letters and count characters.

A Two-Way Case Filter Program

The following program can perform two distinct functions: It can force all lowercase characters in a text file to upper case, or all uppercase characters to lower case. Which of the two actions is taken depends on a parameter entered on the command line:

```
CASE UP B:COBOL.SRC            Forces lower to upper

CASE DOWN B:PASTEXT.SRC        Forces upper to lower
```

Note that the name of the program source *file* is "CASE.PAS," although the name of the *program* is "Caser." The reason is that **CASE** is a reserved word and may not be used as a programmer-defined identifier within a program. However, you may *name* a program code *file* anything you like, and, in this case (so to speak), **CASE** is the best name for the actual runnable program file on disk.

```
{--------------------------------------------------------------}
{                             Case                             }
{                                                              }
{ An upper/lower case conversion filter program for text files }
{                                                              }
{                              by Jeff Duntemann               }
{                              Turbo Pascal V5.0               }
{                              Last update 7/12/88             }
{                                                              }
{                                                              }
{                                                              }
{--------------------------------------------------------------}


PROGRAM Caser;            { "CASE" is a reserved word... }

CONST
  Upper = True;
  Lower = False;

TYPE
  String40   = String[40];
  String80   = String[80];
  String255  = String[255];

VAR
  I,J,K      : Integer;
  Quit       : Boolean;
  Ch         : Char;
  WorkFile   : Text;
  TempFile   : Text;
  NewCase    : Boolean;
  WorkLine   : String80;
  WorkName   : String80;
  TempName   : String80;
  CaseTag    : String80;


{$I FRCECASE.SRC }      { Described in Section 15.3 }


{>>>>MakeTemp<<<<}

PROCEDURE MakeTemp(FileName : String80; VAR TempName : String80);

VAR
  Point : Integer;

BEGIN
  Point := Pos('.',FileName);
  IF Point > 0 THEN Delete(FileName,Point,(Length(FileName)-Point)+1);
  TempName := Concat(FileName,'.$$$')
END;


{ CASER MAIN }

BEGIN
  Quit := False;
  IF ParamCount < 2 THEN    { Missing parms error }
    BEGIN
      Writeln('>>CASE<<  V2.00  By Jeff Duntemann');
      Writeln('          From the book, COMPLETE TURBO PASCAL 5.0');
      Writeln('          Scott, Foresman & Co. 1988');
      Writeln('          ISBN 0-673-38355-5');
      Writeln;
      Writeln('This program forces all characters of a text file to either ');
      Writeln('upper or lower case, as requested.  Characters already in ');
      Writeln('the requested case are not disturbed.');
      Writeln;
      Writeln('CALLING SYNTAX:');
      Writeln;
      Writeln('CASE UP|DOWN <filespec>');
      Writeln;
      Writeln('For example, to force all lowercase characters of file');
      Writeln('FOO.COB to uppercase, invoke CASE this way:');
      Writeln;
      Writeln('CASE UP FOO.COB');
      Writeln;
    END
  ELSE
    BEGIN
      WorkName := ParamStr(2);
      Assign(WorkFile,WorkName);  { Attempt to open the file }
      {$I-} Reset(WorkFile); {$I+}
      IF IOResult <>0 THEN
        BEGIN
          Writeln('<<Error!>> File ',WorkName,' does not exist.');
          Writeln('           Invoke CASE again with an existing FileName.');
          END
      ELSE
        BEGIN                  { See if UP/DOWN parm was entered }
          CaseTag := ParamStr(1);
          CaseTag := ForceCase(Upper,CaseTag);
          IF CaseTag = 'UP' THEN NewCase := Upper ELSE
            IF CaseTag = 'DOWN' THEN NewCase := Lower ELSE
              Quit := True;
          IF Quit THEN
            BEGIN
              Writeln
              ('<<Error!>> The case parameter must be "UP" or "DOWN."');
              Writeln
              ('           Invoke CASE again using either "UP" or "DOWN".');
            END
          ELSE
            BEGIN
              Write('Forcing case ');
              IF NewCase THEN Write('up ') ELSE Write('down ');
```

```
                    MakeTemp(WorkName,TempName);  { Generate temporary FileName }
                    Assign(TempFile,TempName);    { Open temporary file }
                    Rewrite(TempFile);
                    WHILE NOT EOF(WorkFile) DO
                      BEGIN
                        Readln(WorkFile,WorkLine);
                        Write('.');               { Dot shows it's working }
                        WorkLine := ForceCase(NewCase,WorkLine);
                        Writeln(TempFile,WorkLine)
                      END;
                    Close(TempFile);              { Close the temporary file }
                    Close(WorkFile);              { Close original source file... }
                    Erase(WorkFile);              { ...and delete it. }
                    Rename(TempFile,WorkName);    { Temporary file becomes source }
                  END
            END
        END
  END.
```

Most of **Caser** is actually setup: making sure files exist; making sure valid commands were entered at the command line, and so on. The real meat of the program is simplicity itself:

```
WHILE NOT EOF(WorkFile) DO
  BEGIN
    Readln(WorkFile,WorkLine);
    Write('.');                      { Dot shows it's working }
    SetCase(ForceCase,WorkLine);
    Writeln(TempFile,WorkLine);
  END;
```

This loop executes repeatedly as long as there are lines to be read in **WorkFile**. A line is read, **SetCase** adjusts the case of the characters in the line, and then the line is written to **TempFile**. When **Readln** reads the last line in the text file, the **EOF** function will immediately return **True**. The text file is "filtered" through **SetCase** into a temporary file. When the original file has been read completely, it is erased with **Erase** and the temporary file is renamed to become the original file.

If you're nervous about deleting your original text file (and that is not a totally unhealthful feeling) you could close it with **Close** instead of **Erase** and then give **TempFile** a new file extension instead of .$$$. (I have used ".ZZZ" in the past.) If you're careless about backing up important files this is a very good idea.

19.8: USING BINARY FILES

A text file is distinctive in that it is the *only* type of Pascal file that may contain records of varying lengths. Pascal treats text files specially in other ways, by limiting the range of characters that may legally reside in the file and by the EOF character that accurately flags where written data end. Binary files, by contrast, are given none of this special treatment.

A binary file is a file containing some number of data items of a given type:

```
TYPE
  KeyFile = FILE OF KeyRec;
  CfgFile = FILE OF CfgRec;
  IntFile = FILE OF Integer;
```

Only one data type may be stored in a binary file. You could not, for example, write one variable of type **KeyRec** and another variable of type **CfgRec** to the same logical file.

Each instance of a data item in a binary file is called a "record," whether the data item is a Pascal record type or not. One integer stored in an **IntFile** as defined above could be considered a record of the **IntFile**.

One common use of binary files puts only one record in the file: the *configuration file*. Consider a complicated program that performs file maintenance and telecommunications. The program is used at a great many sites owned by a large corporation. The names of the files it works with change from site to site, as do the telephone numbers it must call to link with the host mainframe computer.

Rather than hard-code things like telephone numbers into the Pascal source file, it makes more sense to store them out to a file. The easiest way is to define a record type containing fields for all the site-specific information:

```
TYPE
  CfgRec  = RECORD
              SiteName   : String;
              SiteCode   : Integer;
              AuthOp     : String;
              HostCode   : Integer;
              PhoneNum   : String;
              P1FileName : String;
              P2FileName : String;
              AXFileName : String
            END;

  CfgFile = FILE OF CfgRec;

VAR
  SiteFile : CfgFile;
  CfgData  : CfgRec;
```

All the data items that change from site to site are present in one single record. When loaded with the correct site values for a particular site, the record can be easily written to disk:

```
Assign(SiteFile,'B:SITEDATA.CFG');
Rewrite(SiteFile);
Write(SiteFile,CfgData);
Close(SiteFile);
```

Here, once **SiteFile** is opened, the single **CfgRec** is written to disk with the **Write** statement.

19.9: USING RANDOM-ACCESS FILES

All file access methods we have discussed so far have been *sequential*. That is, when you open a file, you may access the first record, then the second, then the third, and so on until you run out of records. All records in the file are accessible, but only at the cost of always starting at the beginning and scanning past all records up to the one you need.

"Random access" of a file means the ability to open a file and simply "*zap!*" read record #241, without having to read anything else first. Then, without any further scanning, simply "*zap*" write data to record #73. Turbo Pascal gives you this ability through the **Seek** procedure.

Random access is possible with all binary files. Text files may *not* be accessed randomly; for random access to work, all records in a file must be the same length. Binary files that were written sequentially with **Write** may be read or rewritten randomly by using **Seek**. Binary files written randomly by using **Seek** may be read sequentially with **Read**.

Seek is not part of ISO Standard Pascal, although several other Pascals including UCSD Pascal implement it much the same way. **Seek** is predeclared this way:

```
PROCEDURE Seek(FileVar : <binary filetype>; RecNum : LongInt);
```

Seek manipulates the file pointer of opened binary file **FileVar**. **Seek** may *not* be used with text files! It sets the file pointer of **FileVar** to **RecNum**. The next **Read** done on **FileVar** will read the **RecNum**'th record from the file:

```
VAR
  Keys : KeyFile;
  AKey : KeyRec;

Assign(Keys,'NAMES.KEY');
Reset(Keys);

Seek(Keys,17);
Read(Keys,AKey)
```

In this example, record 17 of the file **NAMES.KEY** is read from disk and assigned to the variable **AKey**.

You should be careful not to **Seek** past the end of the file. I/O error #91: Seek beyond end-of-file will result. Testing **FileSize** before **Seek**ing is always a good idea.

Turbo Pascal 3.0 and earlier versions did not have type **LongInt** and used integer variables to specify record positions for small files. A separate procedure requiring a parameter of type **Real**, **LongSeek**, was used for files with more than 32K records. **LongSeek** is still available in the **Turbo3** compatibility unit, but you should convert any code using it to the cleaner and faster **Seek** using long integers.

A Binary Search Procedure

Finding a particular record in a file is perhaps the central problem of all business-related computer science. If you know the record number there is no problem; you either go directly to the record with **Seek** or scan sequentially through the file with **Read**, counting records up to the desired number. But in most cases, you want to locate a record *based on what's in the record.*

A sequential search is simple enough: You start reading at the beginning of the file, and test each record to see if it's the one you want. If it's a big file, or if your search criteria are complicated enough, such a search can take minutes or even hours on hard-disk based systems. Since such a search ties up the computer completely while the search is underway, it can become a costly way to work.

There are many ways to approach the searching of files, but no method is as easy to understand and use as that of binary search.

Binary searches depend upon the file being sorted on the data you wish to search for. In other words, if you want to find a person's name in a file, the records in the file, each containing someone's name, must be in alphabetical order by the name field.

We presented a couple of very fast sorting methods in earlier sections. The Shell sort and quicksort can both be modified to sort any type of data structure that can exist in an array. Sorting a file involves loading its records from disk into an array, sorting the array, and writing the sorted records back out to the file on disk. The file can then be binary searched.

Briefly, binary searching involves dividing a file in half repeatedly, making sure that the desired record is somewhere in the half that is retained at each division. In time the halves divide down to nothing, and if the record is not found at that point it does not exist in the file.

In detail:

The binary search procedure is passed a file to search (called a key file for reasons we'll explain shortly); the number of records in the file; and a data item (in our example, a string) containing the data we're searching for.

Starting out, a variable **Low** contains the record number of the first record in the file (1) and a variable **High** contains the record number of the last record in the file. **High** and **Low** are always the bounds of the region we will be searching. At the outset, they encompass the entire file.

The search begins: The procedure calculates a record number halfway between **High** and **Low** and stores that in **Mid**. The record at **Mid** is read and tested. If the data part of the record at **Mid** matches our "key," the search is over. It probably won't happen quite so quickly.

Because the file is sorted, we can state this: If our key is *greater* than the data at **Mid**, we must now search the half of the file above **Mid**. If our key is *less* than the data at **Mid**, we must search the half of the file below **Mid**.

The procedure thus sets up a new **High** and **Low** for the half of the file in which our desired record must exist. The process begins again, now with only half of the file. A new **Mid** is calculated, and the record at the new **Mid** is read and tested. If **Mid** isn't the record we want, we have our choice of two new, smaller sections of the file to search. And so we continue, setting up **High** and **Low** as the bounds of a still smaller section of the file.

This continues until one of two things happens: 1) We find that the record we read at **Mid** is the record we want, or 2) **Mid** collides with either **High** or **Low**. If that happens, the search is over without finding what we want. Our key does not exist in the file.

(Of course, if the file is not fully sorted, our desired record may in fact exist in the file and yet the binary search may not find it. The file must be completely and correctly sorted or all bets are off!)

The actual Pascal code for such a binary search function follows. **KeySearch** requires the following type definitions prior to its own definition:

```
TYPE
  KeyRec = RECORD
             Ref     : Integer;
             KeyData : String[25]
           END;

  KeyFile = FILE OF KeyRec;
```

```
{->>>>KeySearch<<<<------------------------------------------}
{                                                            }
{ Filename : KSEARCH.SRC -- Last Modified 7/14/88            }
{                                                            }
{ This routine searches file Keys for key records containing }
{ the key string contained in parameter MatchIt.  The method }
{ is your classic binary search, and the key record type is  }
{ defined as the type show below:                            }
{                                                            }
{     KeyRec = RECORD                                        }
{                Ref     : Integer;                          }
{                KeyData : String30                          }
{              END;                                          }
{                                                            }
{ The function returns True if a matching record is found,   }
{ else False.                                                }
{                                                            }
{                                                            }
{                                                           }
{-----------------------------------------------------------}

FUNCTION KeySearch(VAR Keys     : KeyFile;
                   VAR KeyRef   : Integer;
                       MatchIt  : String80) : Boolean;

VAR High,Low,Mid : Integer;
    SearchRec    : KeyRec;
    Found        : Boolean;
    Collided     : Boolean;
    RecCount     : Integer;

BEGIN
  KeyRef := 0;                  { Initialize variables      }
  RecCount := FileSize(Keys);
  High := RecCount;
  Low := 0;
  KeySearch := False; Found := False; Collided := False;
  Mid := (Low + High) DIV 2;    { Calc first midpoint       }

  IF RecCount > 0 THEN          { Don't search if file empty}
    REPEAT
      Seek(Keys,Mid);           { Read midpoint record      }
      Read(Keys,SearchRec);
      { Collision between Mid & Low or Mid & High?   }
      IF (Low = Mid) OR (High = Mid) THEN Collided := True;
      IF MatchIt = SearchRec.KeyData THEN  { Found it! }
        BEGIN
          Found := True;            { Set found flag...     }
          KeySearch := True;        { ...function value...  }
          KeyRef := SearchRec.Ref   { ...and file key       }
        END
      ELSE             { No luck...divide & try again  }
        BEGIN
          IF MatchIt > SearchRec.KeyData THEN Low := Mid
            ELSE High := Mid; { Halve the field  }
          Mid := (Low + High + 1) DIV 2;     { Recalc midpoint }
          KeyRef := Mid { Save Mid in parm }
        END
    UNTIL Collided OR Found
END;
```

Keyed Files

KeySearch has some machinery in it that goes beyond simply searching a file for a matching data string. **KeySearch** returns an integer parameter **KeyRef**, which it takes from the **KeyRec** record it locates. The **Ref** field of the **KeyRec** type allows us to build a "keyed" file system.

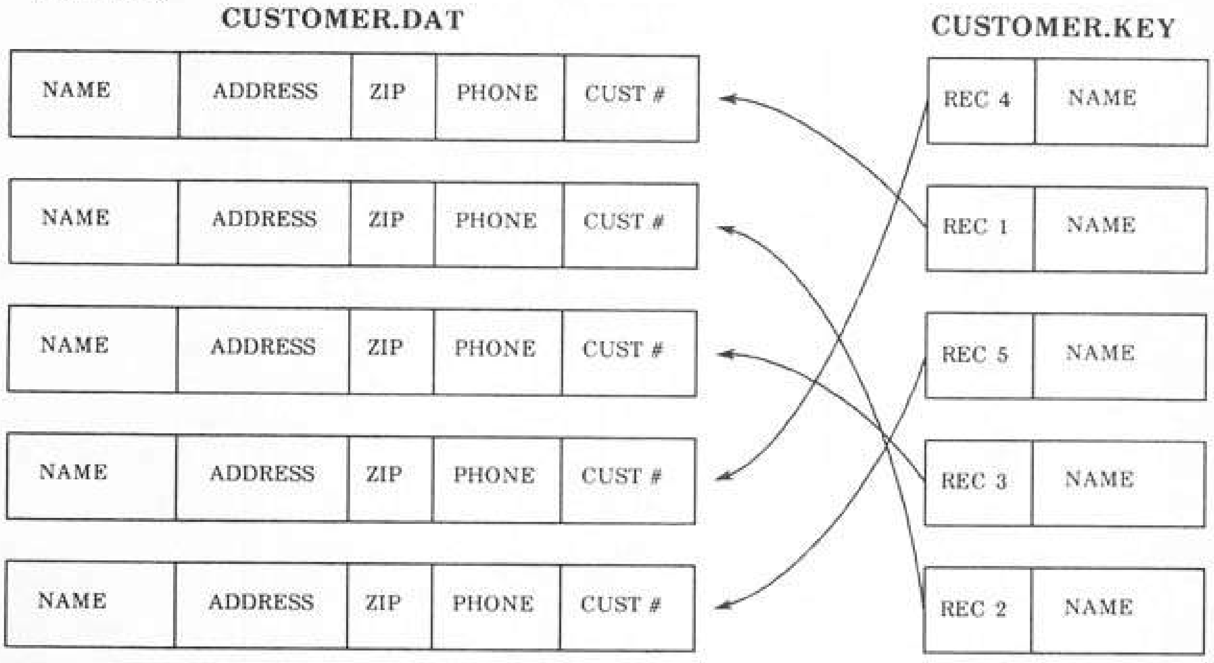
It is both difficult and hazardous to sort a large data file composed of large ("wide") records. It is difficult because the entire file (or big chunks of it for a sort/merge system) must be in memory at one time; and it is hazardous because sorting involves rewriting the entire file after every sort. Rewriting an entire file greatly increases vulnerability to disk errors that can corrupt data in the file, or in the worst case even make the file unreadable.

Furthermore, sorting a file involves swapping many large records around, even if the part of the record that is sorted (called the "key field") is a very small part of the entire record. A lot of that swapping time is simply wasted.

It would be better, faster, and safer to extract and sort only that part of the record that needs to be sorted. This is what a key file is for. The **KeyData** field is that data that is sorted. The **Ref** field is the record number of the record in the main data file from which the **KeyData** field was extracted. If we binary search a key file for a given string, **Ref** will give us the record number where we can find the rest of the data associated with that string. In other words, if we binary search a key file containing only names, **Ref** will allow us to read the record containing the address, phone number, and other information associated with the name we found (see Figure 19.1).

Figure 19.1

Keyed Files



Records in the data file are stored in the order they were entered. The data file itself is never sorted. Instead, keys are extracted from the data file into a key file. Then the smaller key file is sorted. This also presents less risk of damaging the data file.

Key files contain fields extracted from the data file, plus pointers into the data file. Records are in sort order by the key field.

The following simple program assumes the existence of a data file and key file sorted on the data file's **Name** field. (Two such files are provided on the listings diskette for this book.) The program waits for a name to be entered, then searches the key file for the name. If the name is present in the key file, the program reads the data record for the rest of the data associated with that name, and displays it.

```
{---------------------------------------------------------------}
{                          ShowName                             }
{                                                               }
{           Keyed file binary search demo program               }
{                                                               }
{                             by Jeff Duntemann                 }
{                             Turbo Pascal V5.0                 }
{                             Last update 7/24/88               }
{                                                               }
{                                                               }
{                                                               }
{---------------------------------------------------------------}

{ Unlike most programs in this book, this program requires two }
{ external files to operate: FRIENDS.NAP and FRIENDS.KEY.  The }
{ two files will be included on the source listings diskette.  }
{ FRIENDS.NAP is a file of NAPRec containing some number of    }
{ name/address/phone records.  FRIENDS.KEY is a sorted key     }
{ file containing keys extracted from FRIENDS.NAP.  You can    }
{ write a utility to extract keys from a .NAP file and sort    }
{ them using either the SHELLSORT or QUIKSORT procedures given }
{ in Section 14. }

PROGRAM ShowName;

TYPE
  String3      = String[3];
  String6      = String[6];
  String30     = String[30];
  String40     = String[40];
  String80     = String[80];
  String255    = String[255];

  NAPRec = RECORD
             Name    : String30;
             Address : String30;
             City    : String30;
             State   : String3;
             Zip     : String6
           END;

  NAPFile = FILE OF NAPRec;

  KeyRec  = RECORD
              REF : Integer;
              KeyData : String30
            END;

  KeyFile = FILE OF KeyRec;

```

```

VAR I,J,K      : Integer;
    RecNum     : Integer;
    Parm       : String80;
    WorkRec    : NAPRec;
    WorkFile   : NAPFile;
    WorkKey    : KeyFile;


{$I KSEARCH.SRC}   { Contains KeySearch }


{ SHOWNAME MAIN }

BEGIN
  IF ParamCount < 1 THEN            { Missing parms error }
    BEGIN
      Writeln('<<Error!>> You must enter a name on the command line:');
      Writeln('           A>SHOWNAME Duntemann*Jeff ')
    END
  ELSE
    BEGIN
      Parm := ParamStr(1);
      Assign(WorkFile,'FRIENDS.NAP');  { Open the names data file }
      Reset(WorkFile);
      Assign(WorkKey,'FRIENDS.KEY');   { Open the names key file }
      Reset(WorkKey);
      IF KeySearch(WorkKey,RecNum,Parm) THEN  { If key is found...}
        BEGIN                          { We have record # into data file }
          Seek(WorkFile,RecNum);       { Seek to record # in data file }
          Read(WorkFile,WorkRec);      { Read data record from data file }
          WITH WorkRec DO              { and display the name/address data }
            BEGIN
              Writeln('>>NAME    : ',Name);
              Writeln('  ADDRESS : ',Address);
              Writeln('  CITY    : ',City);
              Writeln('  STATE   : ',Zip);
            END
        END
      ELSE
        Writeln('>>Sorry, ',Parm,' not found.');
    END
END.
```

Records are added to the data file at end of the file, and are never rewritten unless the data must be changed somehow.

Another important advantage of a keyed file system is that your main data file can effectively be sorted on two or more of its fields at the same time. You can just as easily have a key file keyed on the **Address** or **Zipcode** field. There is the disk space overhead required for the additional key files, but that is nothing like the space it would take to hold the same data file duplicated in its entirety once for each field!

19.10: USING UNTYPED FILES

All the file access methods we've discussed so far make some kind of assumption about the structure of the file being read or written. A file is really nothing more than a collection of sectors on a disk containing bytes of information. Turbo Pascal allows you to treat a file as a series of blocks without any assumptions about what type of data the file contains.

Any disk file may be declared as untyped and accessed on a block-by-block basis. The declaration for an untyped file simply omits a type for the file:

```
WorkFile : FILE;
```

It doesn't matter how the file was originally written. Text files, binary files, program files, any files, whether created by Turbo Pascal or any other program, may be declared and opened as an untyped file.

Untyped files have a default record length of 128 bytes. This means that whenever you perform a **BlockRead** or a **BlockWrite** on an untyped file (see below) 128 bytes of data are moved between the disk and the file buffer or vise versa.

This is handy for programs which need to process data in 128 byte chunks, like the **HexDump** program we'll look at shortly. Often it is better to be able to move data in larger blocks. Turbo Pascal allows you to set the block size for untyped files by an optional parameter on the **Reset** and **Rewrite** statements:

```
PROCEDURE Reset(FileName : String; BlockSize : Word);
PROCEDURE Rewrite(FileName : String; BlockSize : Word);
```

For both procedures, **BlockSize** must be 128 or a multiple of 128.

Note that using the optional block size parameter with a typed file will generate:

```
Error 89: ')' expected
```

You can use the block size parameter *only* with untyped files!

Accessing untyped files is done with two routines: **BlockRead** and **BlockWrite**. They are virtually identical except for the direction data move:

```
PROCEDURE BlockRead(VAR AFile : FILE;
                    VAR Buffer;          { Untyped VAR parm! }
                    Blocks : Word;
                    Result : Word);      { This parm optional }

PROCEDURE BlockWrite(VAR AFile : FILE;
                     VAR Buffer;         { Untyped VAR parm! }
                     Blocks : Word;
                     Result : Word);     { This parm optional }
```

Parameter **AFile** is an untyped file. **Buffer** is an untyped **VAR** parameter (see Section 23.3) and may therefore be a variable of any type, provided that it is at least 128 bytes or a multiple of 128 bytes in size. **Blocks** is the number of blocks to be read or written. These blocks will default to 128 bytes in size, or their size can be set with an optional parameter to **Reset** and **Rewrite**, as described above. **Result** indicates the number of blocks that were actually read or written. **Result** is an optional parameter; you may omit it if you like.

I/O on untyped files is very fast because the data need not be sorted out into lines or structures; they are simply moved *en masse* between the disk and **Buffer**. It is most useful for file moves and for those applications in which data in a file are best treated as large lumps without interpretation.

One simple example is the saving and loading of graphics screens on the IBM PC Color Graphics Adapter. Graphics images exist in a 16384-byte RAM buffer on the adapter. Saving a graphics screen is as easy as saving a memory image of the buffer to disk:

```
{<<<< GSave >>>>}
{                                                    }
{                                                    }
{                                                    }

PROCEDURE GSave(GName : String; VAR IOR : Integer);

TYPE
  ScreenBuff = ARRAY[0..16383] OF Byte;

VAR
  GBuff : ScreenBuff ABSOLUTE $B800 : 0;
  GFile : File;

BEGIN
  Assign(GFile,GName);
  Rewrite(GFile,16384);
  BlockWrite(GFile,GBuff,1);
  IOR := IOResult;
  Close(GFile)
END;
```

GBuff is an *absolute* variable; that is, a variable that is explicitly created at a particular location in memory, usually outside the program's data area (in this case at $B800 : $0, the address of the Color Graphics Adapter's video refresh buffer see Section 23.2).

Loading a saved graphics image from disk into the Color Graphics Adapter buffer is just as easy:

```
{<<<< GLoad >>>>}
{                                                    }
{                                                    }
{                                                          }

PROCEDURE GLoad(GName : String; VAR IOR : Integer);

TYPE
  ScreenBuff = ARRAY[0..16383] OF Byte;

VAR
  GBuff : ScreenBuff ABSOLUTE $B800 : 0;
  GFile : File;

BEGIN
  Assign(GFile,GName);
  {$I-} Reset(GFile,16384); {$I+}
  IOR := IOResult;
  IF IOR = 0 THEN
    BEGIN
      BlockRead(GFile,GBuff,1);
      Close(GFile)
    END
END;
```

For both procedures, the **GName** parameter is a string containing the full filename of the graphics file to be read or written. **IOR** returns the **IOResult** I/O status code from the block I/O operation, which may be tested by the calling logic to determine how the operation went, and take action accordingly. Note that the block size is set to 16,384, the size of a RAM buffer on the Color Graphics Adapter. This means that one block I/O operation will transfer an entire screen between disk and video buffer. This is the fastest way within Turbo Pascal of moving a graphics screen between disk and buffer.

The **GraphFiler** program below demonstrates the speed of **BlockRead** and **BlockWrite** by drawing a graphics image on the screen, saving it to disk through the **GSave** procedure described above, and then reading it back from disk through **GLoad**. The **GraphFiler** program creates a 16,384-byte file in the current directory; if you're using diskettes, make sure you have room for the file!

```
{--------------------------------------------------------------}
{                          GraphFiler                          }
{                                                              }
{            Graphics file I/O demonstration program           }
{                                                              }
{                             by Jeff Duntemann                }
{                             Turbo Pascal V5.0                }
{                             Last update 7/24/88              }
{                                                              }
{                                                              }
{                                                              }
{--------------------------------------------------------------}

PROGRAM GraphFiler;

USES Crt,Graph3;    { Uses Turbo Pascal 3.0-style graphics for }
```

```
                     { simplicity's sake... }

VAR
  I         : Integer;
  ErrorCode : Integer;


{$I GSAVE.SRC}
{$I GLOAD.SRC}


BEGIN  { GraphFiler MAIN }
  ClrScr;                           { Clear the text screen }
  HiResColor(15);                   { Use white as foreground color }
  HiRes;                            { Clears graphics screen }
  TextColor(1);
  FOR I := 0 TO 199 DO              { Draw lines }
    IF I MOD 5 = 0 THEN Draw(0,0,640,I,1);
  GSave('LINES.PIC',ErrorCode);     { Save graphics image to a file }
  Write('Press RETURN to clear screeen and re-load image: ');
  Readln;
  HiRes;                            { Clears graphics screen }
  GLoad('LINES.PIC',ErrorCode);     { Load saved file into display buffer }
  Readln                            { Wait for RETURN before terminating }
END.
```

A Hex Dump Program

GSave and **GLoad** treat an area of memory as one big block. Their whole purpose is to read and write a large memory image to or from disk as quickly as possible. Other applications may involve reading a file as many small blocks and working with the data one block at a time.

The following program displays a hex dump of any disk file:

```
{--------------------------------------------------------------}
{                           HexDump                            }
{                                                              }
{             Hex dump program for all disk files              }
{                                                              }
{                              by Jeff Duntemann               }
{                              Turbo Pascal V5.0               }
{                              Last update 7/14/88             }
{                                                              }
{                                                              }
{                                                              }
{--------------------------------------------------------------}

PROGRAM HexDump;

{$V-}  { Relaxes String length type checking on VAR paramaters }

CONST
  Up    = True;
  Down = False;

TYPE
  String255   = String[255];
  String128   = String[128];
  String80    = String[80];
  String40    = String[40];
  Block       = ARRAY[0..127] OF Byte;  { One disk sector   }
  BlockArray  = ARRAY[0..15] OF Block;  { BlockRead reads   }
                                        { 16 Blocks at once }


VAR
  I,J,K       : Integer;
  Parm        : String80;
  Ch          : Char;
  DumpFile    : FILE;
  XBlock      : Block;
  DiskData    : BlockArray;
  Blocks      : Integer;        { Counts Blocks within }
                                { BlockArray }
  BlockCount  : Integer;        { Tallies total # Blocks Read }
  Buffers     : Integer;
  Remains     : Integer;
  Device      : Text;           { Will be either LST: or CON: }
  BytesRead   : Integer;


{$I FRCECASE.SRC }   { Described in Section 15.3 }
{$I YES.SRC }        { Described in Section 18.3 }
{$I WRITEHEX.SRC }   { Described in Section 23.5 }


{>>>>DumpBlock<<<<}

PROCEDURE DumpBlock(XBlock : Block; VAR Device : Text);

VAR
  I,J,K : Integer;
  Ch    : Char;

BEGIN
  FOR I:=0 TO 7 DO        { Do a hexdump of 8 lines of 16 chars }
    BEGIN
      FOR J:=0 TO 15 DO   { Show hex values }
        BEGIN
          WriteHex(Device,Ord(XBlock[(I*16)+J]));
          Write(Device,' ')
        END;
      Write(Device,'   |');    { Bar to separate hex & ASCII }
      FOR J:=0 TO 15 DO        { Show printable chars or '.' }
        BEGIN
          Ch:=Chr(XBlock[(I*16)+J]);
          IF ((Ord(Ch)<127) AND (Ord(Ch)>31))
          THEN Write(Device,Ch) ELSE Write(Device,'.')
        END;
      Writeln(Device,'|')
    END;
  FOR I:=0 TO 1 DO Writeln(Device,'')
END;  { DumpBlock }


{<<<<ShowHelp>>>>}

PROCEDURE ShowHelp(HelpName : String80);

VAR
  HelpFile : Text;
  HelpLine : String80;
  I        : Integer;

BEGIN
  Writeln;
  Assign(HelpFile,HelpName);
  {$I-} Reset(HelpFile); {$I+}
  IF IOResult = 0 THEN
    FOR I := 1 TO 24 DO
      BEGIN
        Readln(HelpFile,HelpLine);
        Writeln(HelpLine)
      END;
  Close(HelpFile)
END;


BEGIN
  Parm := '';
                                  { Caps lock printer parameter }
  IF ParamCount > 1 THEN Parm := ForceCase(Up,ParamStr(2));
  IF ParamCount < 1 THEN          { Error - no parms given }
    BEGIN
      Writeln('<<Error!>> You must enter a filename after invoking');
      Write  ('           HexDump.COM.  Display help screen? (Y/N): ');
      IF Yes THEN ShowHelp('DUMPHELP.TXT')
    END
  ELSE
    BEGIN
      Assign(DumpFile,ParamStr(1));  { Attempt to open the file }
      {$I-} Reset(DumpFile); {$I+}
      IF IOResult <> 0 THEN        { Error if file won't open }
        BEGIN
          Writeln('<<Error!>> File ',ParamStr(1),' does not exist.');
          Write  ('           Display help screen? (Y/N): ');
          IF Yes THEN ShowHelp('DUMPHELP.TXT');
        END
      ELSE
        BEGIN                          { See if print Parm was entered; }
                                       { and select output Device }
          IF (Pos('PRINT',Parm) = 1) OR (Pos('P',Parm) = 1) THEN
            Assign(Device,'PRN') ELSE Assign(Device,'CON');
          Rewrite(Device);
          BlockCount := FileSize(DumpFile) + 1; { FileSize in 128-Byte Blocks }
          IF BlockCount = 0 THEN
            Writeln('File ',ParamStr(1),' is empty.')
          ELSE
            BEGIN
```

```
              Buffers := BlockCount DIV 16;   { # of 16-Block Buffers }
              Remains := BlockCount MOD 16;   { # of Blocks in last buffer }
              FOR I := 1 TO Buffers DO        { Dump full 16-Block Buffers }
                BEGIN
                  BlockRead(DumpFile,DiskData,16,BytesRead); { Read 16 disk Blocks }
                  FOR J := 0 TO 15 DO
                    DumpBlock(DiskData[J],Device)  { Dump 'em... }
                END;
              IF Remains > 0 THEN  { If fractional buffer Remains, dump it }
                BEGIN
                  BlockRead(DumpFile,DiskData,Remains,BytesRead); { Read last buffer }
                  FOR I := 0 TO Remains-1 DO
                    DumpBlock(DiskData[I],Device)      { Dump it }
                END
            END;
          Close(DumpFile)
        END
    END
END.
```

As with the **CASE** program, parameters are extracted from the command line with **ParamCount** and **ParamStr**. The first parameter is the name of the file to be dumped; the second is optional: The word **PRINT** or the letter **P**, indicating that the dump is to be sent to the printer:

```
A>HEXDUMP B:MYFILE.TXT
A>HEXDUMP B:MYFILE.TXT PRINT
A>HEXDUMP B:MYFILE.TXT P
```

The program uses **BlockRead** to read the file 128 bytes at a time, and then display the 128-byte block in hexadecimal format using **WriteHex**. Another interesting feature of this program is a simple on-line help system using a small text file. When an error message is displayed, the user is asked if he or she would like to see the help file; if they answer yes, the text file containing help information is opened and read line by line to the screen.

The text file may be produced on any text editor, and may be anything at all as long as it has no more than 23 lines of 79 or fewer characters. Most CRT systems will linefeed when 80 characters are displayed on a single line, and if an 80-character line has a CR/LF pair at its end, your help screen will be double-spaced and scroll off the top of the screen.

19.11: USING MS/PC DOS STRUCTURED DIRECTORIES

Version 2.0 and later of MS/PC DOS supports "structured directories"; that is, directories that may contain not only files but also subsidiary directories called *subdirectories*. These subdirectories are functionally identical to the "root" directory on a disk, and may contain files or subdirectories of their own.

This is not the place to discuss the details of DOS structured directories. Peter Norton's *MS DOS and PC DOS User's Guide* (Bowie, M. D.: Robert W. Brady Co. 1984) provides one of the better treatments.

Versions 1 and 2 of Turbo Pascal did not contain support for structured directories. Starting from version 3.0, Turbo Pascal supports DOS structured directories. This section describes that support.

GetDir

This procedure allows Turbo Pascal programs to determine what the current directory is on any drisk drive on the system. It is predeclared this way:

```
PROCEDURE GetDir(Drive : Byte; VAR CurrentDirectory : String);
```

The input to **GetDir** is **Drive**, a **Byte** parameter containing a value that specifies which disk drive is to be queried. The correspondence between values passed in **Drive** and physical drive specifiers runs like this:

```
0 = The logged drive
1 = A:
2 = B:
3 = C:
```

and so on.

GetDir's output is **CurrentDirectory**, a string which returns the path of the current directory on the specified disk drive.

The following program will display the current directories for the logged drive and drives A through D:

```
{--------------------------------------------------------------}
{                          ShowDir                             }
{                                                              }
{                  "GetDir" demonstration program              }
{                                                              }
{                              by Jeff Duntemann               }
{                              Turbo Pascal V5.0               }
{                              Last update 7/25/88             }
{                                                              }
{                                                              }
{                                                              }
{--------------------------------------------------------------}

PROGRAM ShowDir;

```

```
VAR
  I     : Byte;
  Error : Integer;
  CurrentDirectory : String;

BEGIN
  FOR I := 0 TO 4 DO
    BEGIN
      GetDir(I,CurrentDirectory);
      IF I = 0 THEN Write('Logged drive: ')
        ELSE Write('Drive      ',Chr(64+I),': ');
      Writeln(CurrentDirectory)
    END
END.
```

There must be a diskette in any diskette drives when **ShowDir** is run, or DOS will generate its familiar, "Abort, Retry, or Ignore?," error. You can press I for Ignore to continue program execution.

One unfortunate weakness of **GetDir** is that it does not return any error condition for checking on a disk drive that doesn't exist; that is, if you have a drive A: and drive C: but no drive B:. **IOResult** returns 0 in all cases. Determining what disk drives actually exist from within Turbo Pascal is not trivial, and I have not worked out a totally reliable way of doing so in all cases.

MkDir

Creating a new subdirectory is the job of **MkDir**, predeclared this way:

```
PROCEDURE MkDir(NewDirectory : String);
```

When invoked, **MkDir** will create a new subdirectory with the path specified in **NewDirectory**.

When you use **MkDir**, *always* disable runtime error checking around it, as you should always do with **Reset** (see Section 19.4) followed by an invocation of the **IOResult** function. Trying to create a subdirectory which already exists, or a subdirectory on a volume that has been marked Read Only, or with an invalid path, will trigger a runtime error and terminate your program unless runtime error checking has been disabled. **IOResult** will return 1 for all ordinary errors, and 0 if the operation completed successfully.

RmDir

Deleting a subdirectory is accomplished with **RmDir**:

```
PROCEDURE RmDir(TargetDirectory : String);
```

The path of the subdirectory to be removed is passed to **RmDir** in **TargetDirectory**. As with **MkDir**, runtime error checking should be disabled around the **RmDir** statement, as runtime errors can occur if the subdirectory does not already exist or if it still contains undeleted files. Again, the returned error code from **IOResult** will be 1. 0 indicates that the subdirectory was successfully removed.

You cannot delete the root directory, the current directory (often indicated by a single period in a pathname: .) or the parent directory (often indicated in a pathname with two periods: ..). If you try to delete what looks like an empty directory and still get an error, there may be files in the directory marked as "hidden" or "system" files, and therefore not displayed from the DOS DIR command. Inspect the subdirectory with a directory utility like **The Norton Utilities**, **Window DOS**, or one of the many good public-domain directory utilities available from your user group. If there are files of any kind in a directory, you cannot delete it!

ChDir

The "current directory" of a disk drive can be simply thought of as the directory named in the DOS prompt with the (essential, as far as I'm concerned) PROMPT PG command in force. In other words, if you're working on drive C: and the command prompt says

```
C:\TURBO\HACKS\GRAPHICS>
```

the current directory for drive C: is \TURBO\HACKS\GRAPHICS. The current directory is where DOS looks for programs and files when no pathname to a specific directory is given.

Even though you may execute a program from a particular directory (say, \TURBO) your program can change the current directory to something entirely different. When it finishes executing, the current directory will remain changed; nothing in Turbo Pascal's runtime code will automatically change it back. If you want to change the current directory back to what it was originally, you must first save the current directory path in a string variable by invoking **GetDir** (see above) and then changing back to the original directory by executing another **ChDir** just before your program terminates.

ChDir is predeclared this way:

```
PROCEDURE ChDir(TargetDirectory : String);
```

Just as with the other procedures that operate with subdirectories, **ChDir** can trigger runtime errors if the specified directory does not exist, or if the given path in **TargetDirectory** is somehow invalid. Turn runtime error checking off around **ChDir** and sample the error condition with **IOResult** after each invocation.

20 PC DOS and the DOS Unit

The best 8086 software toolkit bargain going is PC DOS. For $90.00 you get a host of machine-code subprograms of remarkable power, which taken together, do most of the dullest and most difficult routine work of a PC program, such as telling the time and date, spinning the disks, and managing memory. All of this power is available through the standard Turbo Pascal unit, **DOS**. The most widely used functions have their own high-level procedures and functions within the **DOS** unit that hide much of the arcana connected with loading and unloading machine registers and so on. To access DOS functions not graced with their own high-level subprograms, a generalized DOS-call primitive is provided from which *any* DOS call can be made with a little study and care.

Some of the material in this section encroaches on the advanced, particularly that connected with software interrupts and "naked" DOS calls. Take it slow and easy, and read everything with an eye for the details. If it doesn't become clear immediately, come back to this section once you've worked up some confidence in general Pascal programming.

20.1: THE EVOLUTION OF A TOOLKIT

Those of us who shrug at the thought of an application program that *requires* 640K (like Xerox's superb Ventura Publisher) might find the notion of an operating system living in less than 12K almost miraculous until we ponder how very little such an operating system actually accomplished. The operating system, of course, was CP/M-80, the overwhelmingly favorite personal computer OS before the appearance of the IBM PC. By 1988, CP/M-80 has been almost forgotten, but to forget it is to lose sight of why PC DOS has evolved the way it has.

DOS 1.X was simply a clone of CP/M-80 written in 8086 assembler rather than 8080/Z80 assember. The first 36 function calls (up to function $24, Set Random Record) are virtually identical to CP/M-80's function calls. Although tightly written, CP/M-80 cannot be accused of having been beautifully designed, and its function calls have a haphazard way of dealing with input values, return values, and error codes. Fortunately, the more fundamental DOS calls are hidden by the Turbo Pascal runtime code and the routines in the standard units, and you'll have very little need to call them directly through the **MSDOS** procedure. When you do, pay attention to the documentation and make no assumptions.

When it accessed files, CP/M-80 set up control blocks in memory called FCB's (File Control Blocks). In general terms, an application wrote necessary information into an FCB and then made the CP/M function call. PC DOS 1.X inherited this system for file I/O function calls, and all the file I/O function calls in the first 36 DOS functions require the use of FCBs.

The break between PC DOS 1.X and DOS 2.0 was a fundamental one, far more fundamental than the break between 2.X and 3.X. For Version 2, Microsoft rewrote DOS from scratch, modeling it on the Unix minicomputer operating system developed by AT&T. DOS 2 and 3 may be seen as stripped-down, single user, single-tasking Unix clones, containing most of the genuine Unix innovations without Unix's wretchedly slow performance.

Unix's most important contributions to DOS are subdirectories and their associated pathnames, which allow structure to be imposed on massive linear directories containing hundreds of separate files. The file I/O code within DOS 1.X was too set in its ways to be expanded to support pathnames and subdirectories, so Microsoft created a whole "second set" of file I/O function calls for DOS 2.X. These new calls provide all the functions of the original set and then some and do it in a much cleaner fashion. DOS 2.X hid the FCB's within itself, making interface to files much simpler and more consistent. Rather than being associated with an FCB, a file, when opened, became associated with a "file handle": a 16-bit code number associated with a file control table somewhere inside DOS.

Most of the fooling around connected with file I/O is, again, handled transparently by the Turbo Pascal runtime code, and you needn't study the arcane details of working with file handles. What you need to understand is why there are two DOS function calls for most file operations: The old ones do not understand pathnames or subdirectories; the new ones do. There are subtler reasons for using the newer set of file function calls, but for brevity's sake I'll ask you to take my word for it. In view of the fact that Turbo Pascal 4.0 and 5.0 do not run under DOS 1.X, you have absolutely no reason *not* to use the newer function calls.

A handful of new function calls were added to DOS 3.X. They are arcane to the nines (seeming to point the way to yet more future enhancements rather than being immediately useful) and I will not cover them here.

20.2: 8088 SOFTWARE INTERRUPTS

Most people think that IBM PC DOS is the controlling hand within the IBM PC, and, while largely true, it doesn't credit the assistance of the IBM ROM BIOS (Basic Input/Output System), which does much of the work for PC DOS. Many of the toolkit routines in this book rely on either PC DOS or ROM BIOS calls. In both instances, those calls are accomplished with software interrupts. In this short section I'll review the concept of a software interrupt and how the concept is realized in the 8088.

An interrupt is a tap on the CPU's shoulder, indicating that it must pay attention to something else *now*. Interrupt mechanisms have always been part of microprocessor systems. Until the development of the 8086 and 8088, all interrupts were *hardware interrupts*.

Hardware interrupts work this way: An electrical signal on one pin of the CPU chip causes the CPU logic to save the program counter, flags register, and the code segment register (CS). The CPU is then free to service the request from outside the chip to execute some bit of code unrelated to its ongoing task. Once the request for service is satisfied, the CPU restores the registers it had saved and picks up its ongoing task as though nothing had happened.

With the 8086 family architecture, Intel presented the concept of the *software* interrupt. Software interrupts work exactly the same way as hardware interrupts do, except that the triggering request is a machine instruction (software) rather than an electrical signal on a CPU pin (hardware).

There were a number of reasons for doing this. Given that the only difference between hardware and software interrupts is the means by which the interrupt request is triggered, it is possible to test hardware interrupt service routines before the interrupting hardware is perfected, by using software interrupts to simulate the incomplete external hardware device.

A far more important use of software interrupts is in the devising of a system of standard entry points to system software. When any kind of interrupt happens, the CPU first saves essential registers and then performs a "long jump" to the location of the interrupt service routine somewhere in memory.

The way the CPU locates this interrupt service routine is critical. The 256 8088 interrupts are numbered from 0 to 255. When an interrupt happens, the CPU must receive the number of the requested interrupt. For hardware interrupts, this number comes from the interrupt priority controller chip outside the CPU. For software interrupts, the interrupt number is built into the interrupt instruction. For example, the 8088 instruction:

```
INT 21H
```

will trigger software interrupt $21.

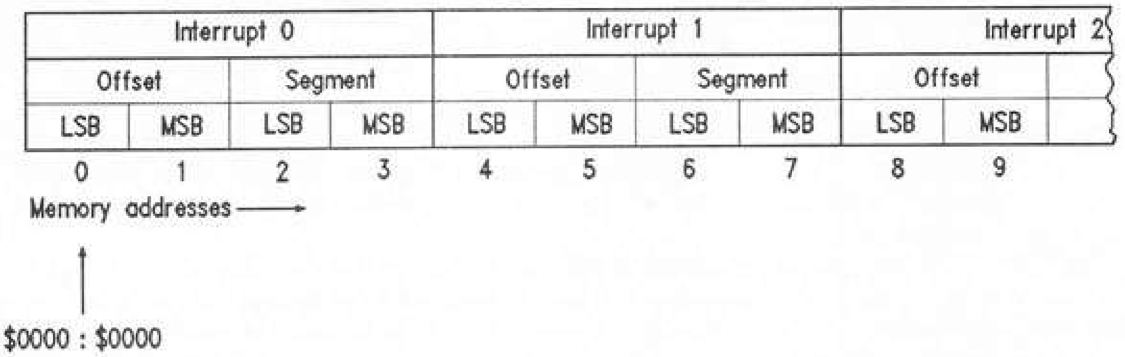
The first 1024 bytes of the 8088 memory map are reserved for interrupt vectors. "Vector" can be taken to mean "pointer," and pointers are exactly what occupy those 1024 bytes of memory. Each of the 256 different interrupts has its own 4-byte region of this 1024-byte memory block ($256 \times 4 = 1024$). This 4-byte region contains a 32-bit pointer to the first instruction of the interrupt's service routine.

The first 4 bytes of 8088 memory contain the interrupt vector for interrupt 0, the next 4 bytes of memory contain the vector for interrupt 1, and so on up to 255 (Figure 20.1). Obviously, if the CPU knows the interrupt number, it can multiply that number by 4 and go immediately to the interrupt vector for any given interrupt. The first 2 bytes of the interrupt vector are the program counter value for the start of the service routine, and the second 2 bytes are the code segment value where that service routine exists. The CPU only needs to load the code segment value into the CS register and the program counter value into the program counter register, and it is off and running the interrupt service routine.

The important fact here is that the code that wishes to use a software interrupt service routine need not know where that routine is in memory. It only needs to know the interrupt number. Indeed, the actual location of the service routine can change over time as the routine is altered or expanded. As long as the computer's boot or startup code stores the correct interrupt vectors into the lowermost 1024 bytes of memory, software

Figure 20.1

The Interrupt Vector Table



interrupt service routines may be anywhere and still be used quickly and easily by application programs.

This is the spirit of the IBM PC's ROM BIOS. The BIOS is a collection of software interrupt service routines stored in a ROM (Read-Only Memory) at the very top of the 8088's memory address space. The interrupt numbers are assigned according to the general function performed by the interrupt service routine. For example, interrupt 16 ($10) controls video services for the PC. Interrupt 22 ($16) controls access to the keyboard.

Not all software interrupts are reserved for the use of the ROM BIOS. PC DOS uses a few, and most of them are not used at all. The Microsoft Mouse driver makes use of a software interrupt. Quite a few peripheral driver programs, in fact, make use of software interrupts. The IBM PC as we know it would have been impossible without them.

Exploring the Interrupt Vector Table with GetIntVec and SetIntVec

Exploring the interrupt vector jump table (the first 1024 bytes of 8086 memory, in which all the vectors are stored) can be done with DEBUG, but a fairly simple program can make the vectors easier to read and change. **VECTORS.PAS** contains a utility called **Vectors** that can display and change interrupt vectors and also display a hex dump of the first 256 bytes of memory pointed to by a vector. Changing an interrupt vector can be strong medicine and shouldn't be attempted unless you know what you're doing. Altering the timer tick interrupt carelessly will freeze your machine solid in no more than 55 milliseconds' time. But if you intend to write programs that intercept interrupt vectors, **Vectors** can save a lot of aggravation during development.

One problem in reading simple hex dumps of the vector table is that the vectors' components are stored backwards in memory from the way we generally write them (see Figure 20.1). The offsets are stored before the segments, and the least significant bytes of both segments and offsets are stored before the most significant bytes.

Vectors reformats the interrupt vector table for our eyes, and allows us to perform certain useful operations on the table. For example, it allows us to zero out all 32 bits of a vector and also to change the offset or segment portion of any vector to the value of our choice.

Vectors turns on two routines from Turbo Pascal's **DOS** unit, **GetIntVec** and **SetIntVec**:

```
PROCEDURE GetIntVec(IntNumber : Byte; VAR Vector : Pointer);

PROCEDURE SetINtVec(IntNumber : Byte; Vector : Pointer);
```

In both cases, **IntNumber** contains the number of the interrupt whose vector you wish to read or change, and **Vector** is a generic pointer containing the address read from or written to that vector. **GetIntVec** returns vector **IntNumber** in **Vector**, and **SetIntVec** places the address in pointer **Vector** in the vector table for interrupt **IntNumber**. These are "well behaved" routines for reading and setting vectors from the interrupt vector table. We say well behaved because the vector table is just an ordinary region of memory, accessible by way of the **Mem**, **MemW**, and **MemL** statements (see Section 23.3). These, however, are considered highly "ill behaved" ways of manipulating interrupt vectors.

Why so?

Well, think about this: Suppose you're in the midst of altering a vector in the table and have half of a new value written, when something somewhere in the system calls that interrupt? At the moment the CPU recognizes the interrupt, you may have a new segment in place but may not have yet overwritten the old offset. The CPU sends execution charging off on this half-baked vector, which lands in the middle of a data buffer, starts executing data as code, and freezes solid.

DOS has a pair of functions for reading and setting interrupt vectors that do it correctly by first *disabling* interrupts before reading or altering a vector. Only when the vector is completely read or completely changed will DOS re-enable interrupts. That way, your programs will not end up reading a half-correct vector or (much worse) allowing the CPU to transfer control to a half-correct vector.

The **Vectors** utility knows another trick. It can provide a look at what any vector is, in fact, pointing to. Any initialized interrupt vector points to an interrupt service routine of some sort. **Vectors** will on command hexdump the first 256 bytes of memory pointed to by any given interrupt vector. People who can read 8086 binary machine code in their heads can track the logic of simple service routines. The rest of us can look for service routine "signatures," typically in the form of copyright notices embedded in the binary machine code. If you have the Logitech mouse driver loaded, **Vectors** will show you the Logitech signature at an offset of 16 bytes into the driver, pointed to by interrupt 51 ($33).

Vectors tests every vector it displays and indicates whether the vector points to a binary $CF value. This is the machine-code equivalent of the IRET (Interrupt Return) mnemonic. Pointing an interrupt to an IRET instruction is a safety measure that prevents havoc in case an interrupt occurs for which the vector is uninitialized. If an unused vector is made to point to an IRET, the worst that can happen if that interrupt is triggered is nothing at all. The IRET sends execution back to the caller without taking any action.

In the best of all worlds, all unused interrupt vectors are initialized to point to an IRET. But as you'll see once you run **Vectors**, only a few vectors are so disarmed. Most point to segment 0, offset 0, which is in fact an interrupt vector itself, the vector for interrupt 0 (the first entry in the jump table). If such an interrupt occurs, the CPU will attempt to execute the interrupt jump table as though it were code, which will almost certainly crash the machine hard.

Vectors is simple in operation. It cycles through the 256 interrupt vectors one at a time, displaying the current value of the current interrupt vector and then pausing for a command. *Jumping* to another interrupt vector is done by entering its value as either a decimal number or a hexadecimal value preceded by a **$**.

Changing a vector value is done with the **E** command. **E** prompts individually for the segment and offset portions of the vector. If you don't wish to change one or both, simply press Enter, and nothing will be altered. As with jumping to a new value, vector values can be entered in either decimal or hex.

D dumps the 256 bytes pointed to by the current vector. If the first byte of the block is an IRET instruction, **Vectors** will say so.

Z changes both the offset and segment portion of a vector to zero. This can be useful in cases where you are testing some software that modifies interrupt vectors, and may be modifying the wrong ones. Zeroing a vector allows you to come back after your test software has run, and tell at a glance if the zeroed vector or vectors have stayed zeroed.

Either **Q** or **X** will exit **Vectors**.

The procedure **WriteHex** figures prominently in **Vectors**, as the mechanism by which the interrupt vectors are displayed, and also the core of a hexdump routine, **DumpBlock**, that dumps 256 bytes of memory at the location pointed to by the current vector.

```
{--------------------------------------------------------------}
{                           VECTORS                            }
{                                                              }
{                   Interrupt vector utility                   }
{                                                              }
{                             by Jeff Duntemann                }
{                             Turbo Pascal V5.0                }
{                             Last update 7/1/88               }
{                                                              }
{ This program allows you to inspect and change 8086 interrupt }
{ vectors, and look at the first 256 bytes pointed to by any   }
{ vector.  This allows the spotting of interrupt service       }
{ routine "signatures" (typically the vendor's copyright       }
{ notice) and also indicates when a vector points to an IRET.  }
{                                                              }
{                                                                }
{                                                                }
{----------------------------------------------------------------}


PROGRAM Vectors;

USES DOS;      { For GetIntVec and SetIntVec }

{$V-}          { Relaxes type checking on string lengths }

CONST
  Up = True;

TYPE
  String80    = String[80];
  Block       = ARRAY[0..255] OF Byte;
  PtrPieces   = ARRAY[0..3] OF Byte;

VAR
  I             : Integer;
  VectorNumber  : Integer;
  Vector        : Pointer;
  VSeg,VOfs     : Integer;
  NewVector     : Integer;
  MemBlock      : Block;
  ErrorPosition : Integer;
  Quit          : Boolean;
  Command       : String80;
  CommandChar   : Char;



PROCEDURE StripWhite(VAR Target : String);

CONST
  Whitespace  : SET OF Char = [#8,#10,#12,#13,' '];

BEGIN
  WHILE (Length(Target) > 0) AND (Target[1] IN Whitespace) DO
    Delete(Target,1,1)
END;


PROCEDURE WriteHex(BT : Byte);

CONST
  HexDigits : ARRAY[0..15] OF Char = '0123456789ABCDEF';

VAR
  BZ : Byte;

BEGIN
  BZ := BT AND $0F;
  BT := BT SHR 4;
  Write(HexDigits[BT],HexDigits[BZ])
END;
```

```

{<<<< ForceCase >>>>}
{                                                      }
{                                                      }
{                                                      }

FUNCTION ForceCase(Up : BOOLEAN; Target : String) : String;

CONST
  Uppercase : SET OF Char = ['A'..'Z'];
  Lowercase : SET OF Char = ['a'..'z'];

VAR
  I : INTEGER;

BEGIN
  IF Up THEN FOR I := 1 TO Length(Target) DO
    IF Target[I] IN Lowercase THEN
      Target[I] := UpCase(Target[I])
    ELSE { NULL }
  ELSE FOR I := 1 TO Length(Target) DO
    IF Target[I] IN Uppercase THEN
      Target[I] := Chr(Ord(Target[I])+32);
  ForceCase := Target
END;



Procedure ValHex(HexString : String;
                 VAR Value : LongInt;
                 VAR ErrCode : Integer);

VAR
  HexDigits  : String;
  Position   : Integer;
  PlaceValue : LongInt;
  TempValue  : LongInt;
  I          : Integer;

BEGIN
  ErrCode := 0; TempValue := 0; PlaceValue := 1;
  HexDigits := '0123456789ABCDEF';
  StripWhite(HexString);    { Get rid of leading whitespace }
  IF Pos('$',HexString) = 1 THEN Delete(Hexstring,1,1);
  HexString := ForceCase(Up,HexString);
  IF (Length(HexString) > 8) THEN ErrCode := 9
    ELSE IF (Length(HexString) < 1) THEN ErrCode := 1
  ELSE
    BEGIN
      FOR I := Length(HexString) DOWNTO 1 DO  { For each character }
        BEGIN
          { The position of the character in the string is its value: }
          Position := Pos(Copy(HexString,I,1),HexDigits) ;
          IF Position = 0 THEN   { If we find an invalid character... }
            BEGIN
              ErrCode := I;      { ...set the error code... }
              Exit               { ...and exit the procedure }
            END;
          { The next line calculates the value of the given digit }
          { and adds it to the cumulative value of the string: }
          TempValue := TempValue + ((Position-1) * PlaceValue);
          PlaceValue := PlaceValue * 16;  { Move to next place }
        END;
      Value := TempValue
    END
END;



PROCEDURE DumpBlock(XBlock : Block);

VAR
  I,J,K : Integer;
  Ch    : Char;

BEGIN
  FOR I:=0 TO 15 DO           { Do a hexdump of 16 lines of 16 chars }
    BEGIN
      FOR J:=0 TO 15 DO       { Show hex values }
        BEGIN
          WriteHex(Ord(XBlock[(I*16)+J]));
          Write(' ')
        END;
      Write('   |');                { Bar to separate hex & ASCII }
      FOR J:=0 TO 15 DO         { Show printable chars or '.' }
        BEGIN
          Ch:=Chr(XBlock[(I*16)+J]);
          IF ((Ord(Ch)<127) AND (Ord(Ch)>31))
          THEN Write(Ch) ELSE Write('.')
        END;
      Writeln('|')
    END;
  FOR I:=0 TO 1 DO Writeln('')
END;  { DumpBlock }


PROCEDURE ShowHelp;

BEGIN
  Writeln;
  Writeln('Press RETURN to advance to the next vector.');
  Writeln;
  Writeln('To display a specific vector, enter the vector number (0-255)');
  Writeln('in decimal or preceded by a "$" for hex, followed by RETURN.');
  Writeln;
  Writeln('Valid commands are:');
  Writeln;
  Writeln('D : Dump the first 256 bytes pointed to by the current vector');
  Writeln('E : Enter a new value (decimal or hex) for the current vector');
  Writeln('H : Display this help message');
  Writeln('Q : Exit VECTORS ');
  Writeln('X : Exit VECTORS ');
  Writeln('Z : Zero segment and offset of the current vector');
  Writeln('? : Display this help message');
  Writeln;
  Writeln
  ('The indicator ">>IRET" means the vector points to an IRET instruction');
  Writeln;
END;



PROCEDURE DisplayVector(VectorNumber : Integer);

VAR
  Bump : Integer;
  Chunks : PtrPieces;
  Vector : Pointer;
  Tester : ^Byte;

BEGIN
  GetIntVec(VectorNumber,Vector);{ Get the vector }
  Tester := Vector;                { Can't dereference untyped pointer }
  Chunks := PtrPieces(Vector);     { Cast Vector onto Chunks }
  Write(VectorNumber : 3,'  $');
  WriteHex(VectorNumber);
  Write('  [');
  WriteHex(Chunks[3]);             { Write out the chunks as hex digits }
  WriteHex(Chunks[2]);
  Write(':');
  WriteHex(Chunks[1]);
  WriteHex(Chunks[0]);
  Write(']');
  IF Tester^ = $CF                 { If vector points to an IRET, say so }
    THEN Write(' >>IRET ')
    ELSE Write('        ');
END;


PROCEDURE DumpTargetData(VectorNumber : Integer);

VAR
  Vector : Pointer;
  Tester : ^Block;

BEGIN
  GetIntVec(VectorNumber,Vector);  { Get the vector }
  Tester := Vector;           { Cast the vector onto a pointer to a block }
  MemBlock := Tester^;        { Copy the target block into MemBlock }
  IF MemBlock[0] = $CF THEN { See if the first byte is an IRET }
    Writeln('Vector points to an IRET.');
  DumpBlock(MemBlock)         { and finally, hexdump the block. }
END;



PROCEDURE ChangeVector(VectorNumber: Integer);

VAR
  Vector : Pointer;
  LongTemp,TempValue : LongInt;
  SegPart,OfsPart : Word;

BEGIN
```

```
    GetIntVec(VectorNumber,Vector);   { Get current value of vector }
    LongTemp := LongInt(Vector);      { Cast Pointer onto LongInt }
    SegPart := LongTemp SHR 16;       { Separate pointer segment from offset }
    OfsPart := LongTemp AND $0000FFFF;  { And keep until changed }
    Write('Enter segment ');
    Write('(RETURN retains current value): ');
    Readln(Command);
    StripWhite(Command);
    IF Length(Command) > 0 THEN { If something other than RETURN was entered }
      BEGIN
        Val(Command,TempValue,ErrorPosition);  { Evaluate as decimal }
        IF ErrorPosition = 0 THEN SegPart := TempValue
          ELSE       { If it's not a valid decimal value, evaluate as hex: }
            BEGIN
              ValHex(Command,TempValue,ErrorPosition);
              IF ErrorPosition = 0 THEN SegPart := TempValue
            END;
        Vector := Ptr(SegPart,OfsPart);  { Reset the vector with any changes }
        SetIntVec(VectorNumber,Vector);
      END;
    DisplayVector(VectorNumber); { Show it to reflect changes to segment part }
    Writeln;
    Write('Enter offset  ');     { Now get an offset }
    Write('(RETURN retains current value): ');
    Readln(Command);
    StripWhite(Command);
    IF Length(Command) > 0 THEN { If something other than RETURN was entered }
      BEGIN
        Val(Command,TempValue,ErrorPosition);  { Evaluate as decimal }
        IF ErrorPosition = 0 THEN OfsPart := TempValue
          ELSE       { If it's not a valid decimal value, evaluate as hex: }
            BEGIN
              ValHex(Command,TempValue,ErrorPosition);
              IF ErrorPosition = 0 THEN OfsPart := TempValue
            END
      END;
    Vector := Ptr(SegPart,OfsPart);  { Finally, reset vector with any change: }
    SetIntVec(VectorNumber,Vector);
  END;




  BEGIN
    Quit := False;
    VectorNumber := 0;
    Writeln('>>VECTORS<<  V2.00 by Jeff Duntemann');
    Writeln('             From the book: COMPLETE TURBO PASCAL 5.0');
    Writeln('             Scott, Foresman & Company, 1988');
    Writeln('             ISBN 0-673-38355-5');
    Writeln;
    ShowHelp;

    REPEAT
      DisplayVector(VectorNumber);   { Show the vector # & address }
      Readln(Command);               { Get a command from the user }
      IF Length(Command) > 0 THEN    { If something was typed:     }
        BEGIN
        { See if a number was typed; if one was, it becomes the current }
        { vector number.  If an error in converting the string to a     }
        { number occurs, Vectors then parses the string as a command.   }
        Val(Command,NewVector,ErrorPosition);
        IF ErrorPosition = 0 THEN VectorNumber := NewVector
          ELSE
            BEGIN
              StripWhite(Command);        { Remove leading whitespace   }
              Command := ForceCase(Up,Command); { Force to upper case   }
              CommandChar := Command[1];  { Isolate first char.   }
              CASE CommandChar OF
                'Q','X' : Quit := True;    { Exit VECTORS }
                'D'     : DumpTargetData(VectorNumber); { Dump data }
                'E'     : ChangeVector(VectorNumber);   { Enter new value }
                'H'     : ShowHelp;
                'Z'     : BEGIN            { Zero the vector }
                            Vector := NIL;   { NIL is 32 zero bits }
                            SetIntVec(VectorNumber,Vector);
                            DisplayVector(VectorNumber);
                            Writeln('zeroed.');
                            VectorNumber := (VectorNumber + 1) MOD 256
                          END;
                '?'     : ShowHelp;
              END {CASE}
            END
      END
    { The following line increments the vector number, rolling over to 0 }
    { if the number would have exceeded 255: }
    ELSE VectorNumber := (VectorNumber + 1) MOD 256
  UNTIL Quit;
END.
```

20.3: REGISTERS AND REGISTER STRUCTURES

In broad strokes, the previous section described the nature and use of software interrupts. Your Turbo Pascal programs can make use of software interrupts quite easily. Understanding how requires some discussion of 8088 machine registers.

A register is nothing more than a storage location inside the CPU chip itself. The 8088 has quite a few of them. Nearly all of them are 16 bits, or 2 bytes, wide. Some of the registers have specific duties to perform in the 8088's execution of its programs. Many are simply convenient places to tuck things away for a moment. How the registers contribute to the 8088 instruction set is too large a topic to cover here in detail. What's important now is to understand how machine registers are used in connection with calling software interrupt service routines from Turbo Pascal.

Most software interrupt service routines like those in the PC's ROM BIOS require input values and return output values when processing is finished. These values are passed back and forth between a Turbo Pascal program and the interrupt service routine in machine registers.

Turbo Pascal's **DOS** unit contains the **Intr** procedure, whose job is to invoke software interrupts. **Intr** requires an interrupt number and a special data structure containing word and byte fields corresponding to most of the 8088's registers:

```
Intr(IntNumber: Word; Regs : Registers);
```

The **Registers** type is defined in the interface part of the **DOS** unit. It is a free-union variant record (see Section 9.4) that maps byte fields corresponding to the general-purpose register halves over registers AX, BX, CX, and DX:

```
Registers = RECORD
              CASE Integer OF
              0 : (AX,BX,CX,DX,BP,SI,DI,DS,ES,Flags : Word);
              1 : (AL,AH,BL,BH,CL,CH,DL,DH : Byte);
            END;
```

Each of the fields of **Registers'** 0 variant (that is, the fields defined on the line following 0:) is named after an 8088 machine register. When **Intr** is called by your program, it copies the 10 values from these 10 fields into each field's corresponding 8088 register. Then, it invokes the software interrupt you requested. When the interrupt service routine returns control to **Intr**, **Intr** copies each of the 10 8088 registers into its corresponding field in **Registers**. The interrupt service routine may or may not have changed any of the 10 machine registers, but, if any *were* changed, the changed values will be available in **Registers** for your program to use.

At first glance, this seems straightforward enough. One persistant problem is that many of the 8088's registers are treated by interrupt service routines not as single 16-bit quantities but as independent 8-bit quantities. In particular, registers AX, BX, CX, and DX are often treated as pairs of 8-bit registers. Within AX are AH and AL (think, "A High" and "A Low"), within BX are BH and BL, and so on for CX and DX. You are often called upon to place separate values in the two halves of a given register.

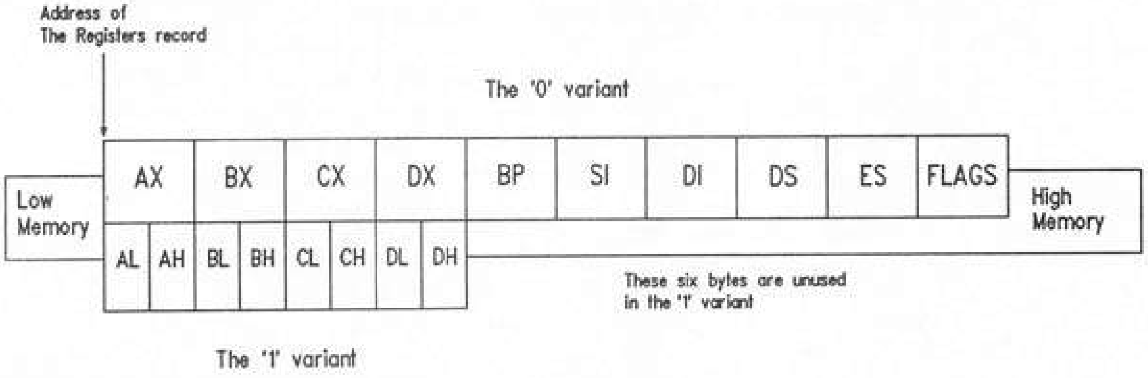
For example, consider the BIOS VIDEO routine that positions the cursor at a particular location on the CRT screen. Before you call software interrupt 16 (VIDEO) you have to place the desired row/column position in register DX. The row number goes in DH and the column number in DL. **Registers** considers DX a word. How do you put two different values into the two halves of a word?

This is what the second variant line is for. Its eight fields are bytes, not words. This means that it takes 2 byte fields to equal the size of one word field, an important characteristic. Consider Figure 20.2, which is a memory diagram of how the different variants of type **Registers** are mapped into memory when Turbo Pascal allocates space for a **Registers** variable.

Both variants begin at the same point in memory, which is the address of the record as a whole. The 0 variant is 10 words long and encompasses everything contained in the record. The 1 variant, by contrast, is only 8 bytes—4 words—long. It too begins at the beginning of the record, but its fields only extend four bytes in, corresponding in memory to the AX, BX, CX, and DX registers of the 0 variant. For example, fields AL

Figure 20.2

The Registers Record



```
TYPE
  Registers = RECORD
                CASE Integer OF
                  0 : (AX,BX,CX,DX,BP,SI,DI,DS,ES,Flags : Word);
                  1 :  (AL,AH,BL,BH,CL,CH,DL,DH : Byte);
              END;
```

and AH of the 1 variant are located at the same memory location as field AX of the 0 variant. This means that whatever exists in that memory location will be copied into the AX register when **Intr** is executed, whether it was written to memory as part of the AX field of the 0 variant, or as part of either the AH or AL fields of the 1 variant.

The **Registers** record type, in a sense, is just a way of casting two **Byte** types onto one **Word** type. Notice that there are no half-registers mapped onto BP, SI, DI, DS, ES, or Flags. In most cases, these registers are not treated by halves. If you need to access half of one of these other registers independently of its other half, you will need to resort to either logical operators (AND or OR) or create a special record type of your own to facilitate the type cast:

```
Halfer = RECORD
           LoHalf,HiHalf : Byte
         END;
```

With type **Halfer** defined, you can cast byte quantities into the correct portion of any word variable:

```
VAR
  Regs : Registers;
```

```
Halfer(Regs.SI).LoHalf := $FF;  { Load the low half of BP }
Halfer(Regs.SI).LoHalf := $1A;  { Load the high half of BP }
```

This notation is *not* standard Pascal and is a Turbo Pascal enhancement to standard Pascal called "variable type casting." See Section 12.3 for more on this particular Turbo trick.

Intr in Action

The best way to show you how to use the **Intr** procedure is with a simple toolkit routine that depends on it. Read the definition of **FlushKey** over carefully:

```
{->>>>FlushKey<<<<------------------------------------------}
{                                                           }
{ Filename: FLUSHKEY.SRC -- Last modified 7/11/88           }
{                                                           }
{ This routine uses ROM BIOS services to flush waiting      }
{ characters from the keyboard buffer.  This should be done }
{ immediately before user prompts which must NOT be answerable }
{ via type-ahead:  Go-aheads for file erasures, things like }
{ that.                                                     }
{                                                           }
{                                                           }
{                                                           }
{-----------------------------------------------------------}

PROCEDURE FlushKey;

VAR
  Regs : Registers;  { USES DOS unit! }

BEGIN
  Regs.AH := $01;             { AH=1: Check for keystroke }
  Intr($16,Regs);             { Interrupt $16: Keyboard services}
  IF (Regs.Flags AND $0040) = 0 THEN  { If chars in buffer }
    REPEAT
      Regs.AH := 0;           { Char is ready; go read it... }
      Intr($16,Regs);         { ...using AH = 0: Read Char }
      Regs.AH := $01;         { Check for another keystroke... }
      Intr($16,Regs);         { ...using AH = 1 }
    UNTIL (Regs.Flags AND $0040) <> 0;
END;
```

FlushKey flushes the keyboard typeahead buffer. In other words, if any characters are waiting to be read in the typeahead buffer, **FlushKey** reads them and throws them away. Why is this useful? There are times when you do *not* want to allow a user to answer a question by typing ahead, presumably before the question's prompt appears on the screen. If you are asking a user whether or not he or she wants to delete the old version of his or her master data file, you would prefer that the user read the question and think about it for a moment rather than insert an answer in the typeahead queue and wait for the machine to catch up with the user's responses. He or she might slip and type the wrong character or forget that the question comes up.

Therefore, immediately before prompting the user for responses that may have drastic consequences, call **FlushKey** to empty the typeahead buffer.

FlushKey uses the keyboard services interrupt of ROM BIOS, interrupt 22 (16 hex, or $16 in the listing). The very first line of the body of **FlushKey** loads a value of 1 into the high byte of register AX:

```
Regs.AH := $01;
```

This notation selects the **AH** field of variant record **Regs** and loads $01 into it. Recall that **AH** indicates a byte-sized field.

Now, consider line 23:

```
IF (Regs.Flags AND $0040) = 0 THEN
```

This time, we're testing for one bit out of the **Flags** register. **Flags** is here treated as a 16-bit word quantity. **Flags** is not tranditionally broken into halves, as AX, BX, CX, and DX so often are. While we could separately access the two halves of **Flags** for bit-testing purposes, I recommend against it for clarity's sake. There is no speed advantage to testing a single bit within an 8-bit byte over testing for that same bit within a 16-bit integer.

A short program to demonstrate **FlushKey** is given below. The idea in **FlushTest** is to perform some time consuming activity (in this case, incrementing a variable to 1000 and printing it all the while) and then query the keyboard to see if this counting should be done again. **FlushKey** is there to clear the keyboard buffer before asking the question. If **FlushKey** were not invoked, the user could type characters into the typeahead buffer and answer the question before it were asked. (Comment out **FlushKey**'s invocation at line 39 and try it!) **Flushkey** guarantees that the user will read and question and (with any luck) think it over before responding.

```
 1  {--------------------------------------------------------------}
 2  {                          FlushTest                           }
 3  {                                                              }
 4  {          Keyboard buffer flush demonstration program         }
 5  {                                                              }
 6  {                            by Jeff Duntemann                 }
 7  {                            Turbo Pascal V5.0                 }
 8  {                            Last update 7/1/88                }
 9  {                                                              }
10  {                                                              }
11  {                                                              }
12  {--------------------------------------------------------------}
13
14
15  PROGRAM FlushTest;
16
17  USES Crt,DOS;
```

```
18
19   VAR I,J : Integer;
20       Ch  : Char;
21
22   {$I FLUSHKEY.SRC}
23
24   PROCEDURE Counter;
25
26   BEGIN
27     FOR I := 1 TO 1000 DO
28       BEGIN
29         GotoXY(1,WhereY); ClrEol;
30         J := J + 1;
31         Write(J)
32       END;
33     Writeln
34   END;
35
36   BEGIN
37     REPEAT
38       Counter;
39       FlushKey;
40       Write('End program now? (Y/N): ');
41       Readln(Ch);
42     UNTIL Ch IN ['Y','y']
43   END.
```

20.4: BIOS AND BIOS SERVICES

FlushKey provides a good introduction to the concept of ROM BIOS services. As I explained earlier, each broad function of BIOS (video, keyboard, printer, serial port) has its own software interrupt. However, within most BIOS functions there are two or more separately invocable services that are called using the same software interrupt. Within the video function (software interrupt 16), for example, are services to position the cursor, write a character to the screen, scroll a rectangular window up or down, change text and graphics modes, write pixels to a graphics screen, read characters from a text screen, and quite a few more.

The BIOS video function is selected by invoking software interrupt 16. Invoking an individual video service is done by placing a service code in the high half of the **AX** register, **AH**. This is a custom in the use of the IBM PC's ROM BIOS services: To select a service within a function, place the service code in **AH**.

FlushKey does this several times, initially at line 21. Service 1 for the BIOS keyboard function checks for the availability of a character in the typeahead buffer. **FlushKey** first checks to see if there is at least one character in the buffer by examining the 0 bit in the **Flags** register. Service 1 communicates its result to the calling logic by setting or clearing the 0 bit. If the 0 bit comes back cleared (i.e., set to 0) a character is in the buffer. If the 0 bit comes back set (i.e., set to 1), no character is waiting in the typeahead buffer.

Notice that if the first check for waiting characters comes back positive, **FlushKey** enters a **REPEAT/UNTIL** loop. The loop is entered only if a character is known to be in the buffer, so keyboard service 0 is called immediately. Service 0 returns the first waiting character in the typeahead buffer. If no character is waiting to be read, service 0 will sit and wait for a character to be typed. This is the reason we must check for the presence of a character before attempting to read one with service 0. Trying to read a character when none has been typed will give the appearance of a hung system until some key is pressed.

The loop reads a character and then checks for the presence of yet another character. If no further characters are detected via keyboard service 1, the buffer is empty and **FlushKey** has done its job. If more characters are detected, the read-and-check-for-more loop is repeated until the buffer comes up empty. This state is indicated by a set Zero flag. Note that nothing is done with the characters read; **FlushKey** simply reads them and throws them away to get them out of the buffer. You could expand **FlushKey** to return a string variable containing any characters found in the typeahead buffer if it were important not to ignore an important keypress, such as an escape code or other "hot key."

Polling the Keyboard with GetKey

With only a little more complication, we can get from the BIOS virtually all it has to offer in terms of keyboard support. There are only three services available under the BIOS keyboard function, interrupt 22. **FlushKey** uses 0 and 1. Service 2 returns the current state of the IBM PC's various shift-type keys.

There are four shift-type keys and four lock-type keys on the PC keyboard. Shift-type keys create a condition only when pressed; when released, that condition goes away. The four shift-type keys are Control (Ctrl), Alt, Left Shift and Right Shift. In terms of uppercase and lowercase characters, Left Shift and Right Shift are identical, but at the BIOS level they are two separate keys that may be tested for independently.

The lock-type keys are Insert (Ins), Caps Lock, Num Lock, and Scroll Lock. These keys control keyboard *toggles*, which are flags that can be switched between two states but which remain in a given state until switched to the opposite state, similar to a toggle switch for a light fixture. Flip it once, and the light goes on and stays on until you flip it again to turn it off. This analogy is especially apt for some non-IBM keyboards, which have little LED lamps built into the lock-type keys so that you can always tell which state the key is in at a glance. With the IBM keyboard you have to remember or take a chance by pressing a key and seeing what happens.

Keyboard interrupt service 2 returns an 8-bit byte in AL containing 8 flag bits. The meaning of each bit is shown in Table 20.1. The states of the shift-type keys (bits 0 through 4) is returned for the moment the interrupt call is made. There is *no* buffering of shift keys.

The routine below is **GetKey**, a keyboard sampling routine which will tell you everything the BIOS is capable of telling you about the IBM PC keyboard. **GetKey** is a

Table 20.1
Meanings of Flag Bits in Keyboard Interrupt Service 2

Bit Number	*Meaning*	*Type*
0	1 = Left Shift depressed	Shift
1	1 = Right Shift depressed	Shift
2	1 = Ctrl depressed	Shift
3	1 = Alt depressed	Shift
4	1 = Scroll Lock active	Lock
5	1 = Num Lock active	Lock
6	1 = Caps Lock active	Lock
7	1 = Insert active	Lock

polling routine. It doesn't wait for a character or a string to be typed at the keyboard. At the moment you call **GetKey**, it jumps out and takes a look at the keyboard and the typeahead buffer. If a character is waiting in the typeahead buffer, **GetKey** will grab it, bring it back, and set its function return value to **True**. If no character is ready, **GetKey**'s function return value will come back as **False**. In *either* case, **GetKey** returns the current state of the shift-type keys and the lock-type keys in a byte named **Shifts**.

```
{->>>>GetKey<<<<--------------------------------------------}
{                                                            }
{ Filename: GETKEY.SRC -- Last modified 7/23/88              }
{                                                            }
{ This routine uses ROM BIOS services to test for the presence }
{ of a character waiting in the keyboard buffer and, if one is }
{ waiting, return it.  The function itself returns a TRUE    }
{ if a character has been read.  The character is returned in }
{ Ch.  If the key pressed was a "special" (non-ASCII) key, the }
{ Boolean variable Extended will be set to TRUE and the scan  }
{ code of the special key will be returned in Scan.  In      }
{ addition, GETKEY returns shift status each time it is called }
{ regardless of whether or not a character was read.  Shift  }
{ status is returned as eight flag bits in byte Shifts,      }
{ according to the bitmap below:                             }
{                                                            }
{             BITS                                           }
{      7  6  5  4  3  2  1  0                                }
{      1  .  .  .  .  .  .  .  INSERT       (1=Active)       }
{      .  1  .  .  .  .  .  .  CAPS LOCK    (1=Active)       }
{      .  .  1  .  .  .  .  .  NUM LOCK     (1=Active)       }
{      .  .  .  1  .  .  .  .  SCROLL LOCK  (1=Active)       }
{      .  .  .  .  1  .  .  .  ALT          (1=Depressed)    }
{      .  .  .  .  .  1  .  .  CTRL         (1=Depressed)    }
{      .  .  .  .  .  .  1  .  LEFT SHIFT   (1=Depressed)    }
{      .  .  .  .  .  .  .  1  RIGHT SHIFT  (1=Depressed)    }
{                                                            }
{ Test for individual bits using masks and the AND operator: }
{                                                            }
{   IF (Shifts AND $0A) = $0A THEN CtrlAndAltArePressed;     }
{                                                            }
{                                                            }
{                                                            }
{------------------------------------------------------------}

FUNCTION GetKey(VAR Ch       : Char;
                VAR Extended : Boolean;
                VAR Scan     : Byte;
                Var Shifts   : Byte) : Boolean;

VAR Regs  : Registers;
    Ready : Boolean;

BEGIN
  Extended := False; Scan := 0;
  Regs.AH := $01;      { AH=1: Check for keystroke }
  Intr($16,Regs);      { Interrupt $16: Keyboard services}
  Ready := (Regs.Flags AND $40) = 0;
  IF Ready THEN
    BEGIN
      Regs.AH := 0;         { Char is ready; go read it... }
      Intr($16,Regs);       { ...using AH = 0: Read Char }
      Ch := Chr(Regs.AL);   { The char is returned in AL }
      Scan := Regs.AH;      { ...and scan code in AH.    }
      IF Ch = Chr(0) THEN Extended := True ELSE Extended := False;
    END;
  Regs.AH := $02;           { AH=2: Get shift/alt/ctrl status }
  Intr($16,Regs);
  Shifts := Regs.AL;
  GetKey := Ready
END;
```

There is an additional complication to the notion of "returning a character" from the PC keyboard. There are keys that do not represent characters of which the function keys and arrow keys are prime examples. Your programs may need to use those keys. How do you get them?

The BIOS divides keystrokes into *ASCII* and *extended* keys. ASCII keys are keys which stand for an ASCII letter, numeral, or symbol having a numeric code of 0 through 127. If an ASCII key is pressed, **GetKey** returns the ASCII character in its character parameter **Ch**. The extended keys are the function keys, arrow keys, PgUp, PgDn, Home, End, and various combinations of those keys and Ctrl, Alt, and Shift. Each extended key has a code from 0 through 255. This code is returned for extended keys in **GetKey**'s **Scan** parameter. If an extended code is being returned, **GetKey** will also return a Boolean value of **True** in its **Extended** parameter.

All the various features of **GetKey** are exercised in the short demo program **KeyTest** given on page 365. **KeyTest** samples the keyboard and displays the last character or extended key pressed, along with the current state of the shift keys and the lock keys. The ASCII character or scan code of the last key pressed will be shown in the middle of the screen. The current state of the shift keys is shown by upward or downward pointing arrows. If a particular lock key is active, the "sunburst" character symbol (character 15) will be displayed next to that key's label. It's interesting to note that of all the shift keys or lock keys, only Insert returns an extended code.

```
{--------------------------------------------------------------}
{                           KeyTest                            }
{                                                              }
{          Full keyboard access demonstration program          }
{                                                              }
{                             by Jeff Duntemann                }
{                             Turbo Pascal V4.0                }
{                             Last update 7/1/88               }
{                                                              }
{                                                              }
{                                                              }
{--------------------------------------------------------------}

PROGRAM KeyTest;

USES Crt,DOS;

VAR Ch : Char;
    Extended : Boolean;
    Scan,Shifts : Byte;
    Ready : Boolean;

{$I FLUSHKEY.SRC }
{$I GETKEY.SRC }
{$I CURSOFF.SRC }

BEGIN
  Ch := ' ';
  Ready := False;
  ClrScr;
  CursorOff;  { Get the cursor out of the way; we don't need it. }
  GotoXY(20,1); Write('< COMPLETE TURBO PASCAL Keyboard Read Demo >');
  GotoXY(30,2); Write('(Press ESC to exit...)');

  { First we set up the labels for the shift keys: }
  GotoXY(12,17); Write('Ctrl: ');
  GotoXY(5,18);  Write('Left Shift: ');
  GotoXY(48,18); Write('Right Shift: ');
  GotoXY(13,19); Write('Alt: ');

  { Here we set up the labels for the toggle keys: }
  GotoXY(50,19); Write('Caps Lock: ');
  GotoXY(64,19); Write('Insert: ');
  GotoXY(52,13); Write('Num Lock: ');
  GotoXY(64,13); Write('Scroll Lock: ');

  GotoXY(31,7); Write('<Last key pressed: >');

  FlushKey; { Empty any waiting keystrokes from the typeahead buffer }
  REPEAT
    Ready := GetKey(Ch,Extended,Scan,Shifts);
    GotoXY(29,8);
    IF Ready THEN  { If a character key has been pressed... }
      IF Extended THEN Write('Extended; Scan code = ',Scan)
```

```
        ELSE Write('           ',Ch,'              ');
    GotoXY(17,18); IF (Shifts AND $02) <> 0 THEN   { Left Shift }
      Write(Chr(31)) ELSE Write(Chr(30));
    GotoXY(62,18); IF (Shifts AND $01) <> 0 THEN   { Right Shift }
      Write(Chr(31)) ELSE Write(Chr(30));
    GotoXY(18,17); IF (Shifts AND $04) <> 0 THEN   { Ctrl }
      Write(Chr(31)) ELSE Write(Chr(30));
    GotoXY(18,19); IF (Shifts AND $08) <> 0 THEN   { Alt }
      Write(Chr(31)) ELSE Write(Chr(30));

    GotoXY(61,19); IF (Shifts AND $40) <> 0 THEN   { Caps Lock }
      Write(Chr(15)) ELSE Write(' ');
    GotoXY(72,19); IF (Shifts AND $80) <> 0 THEN   { Insert }
      Write(Chr(15)) ELSE Write(' ');
    GotoXY(62,13); IF (Shifts AND $20) <> 0 THEN   { Num Lock }
      Write(Chr(15)) ELSE Write(' ');
    GotoXY(77,13); IF (Shifts AND $10) <> 0 THEN   { Scroll Lock }
      Write(Chr(15)) ELSE Write(' ');

  UNTIL Ch = Chr(27);      { Until you press ESC... }
  TextMode(3);             { ...then restore cursor and quit. }
END.
```

Using the PC's Cassette-Control Relay

One of the stranger and today nearly forgotten aspects of the original IBM PC was that it contained a complete digital cassette interface for program and data storage. IBM was covering all bases with its PC; in addition to a fully disk-based machine, it intended to deliver an inexpensive ("only" $2000 or so) version of the PC for use in the home, without any disk drives at all. In place of DOS and disk drives, it had a version of BASICA in ROM that could store and retrieve programs and data from ordinary audio cassettes.

Nobody had told IBM that there was no home computer market for $2000 computers, and the cassette port went generally unused. The cassette port remains on all true-blue IBM PC motherboards, however, and, although the data transfer logic is of little use, the small relay that turned the cassette drive motor on and off can be used for other purposes. Keep in mind that I mean PC here in the specific; the cassette port is *not* part of the PC/XT, PC/AT, or PS/2 machines.

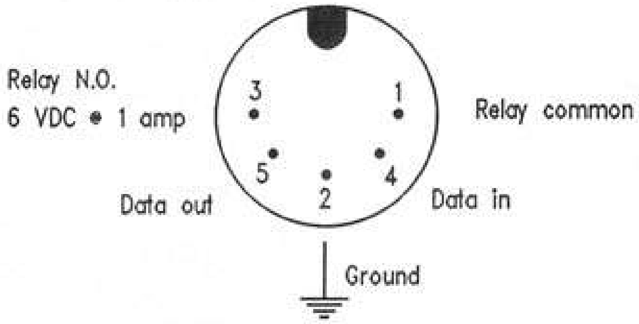
The relay is an SPST type, with contacts rated at one amp current. Having taken a long look at the relay, however, I would play it safe and limit the current through the relay to half an amp or only a little more. The relay contacts are brought out through that other five-pin DIN connector on the back panel of the PC, into which you are always connecting the PC keyboard by mistake. The DIN connector pinouts are shown in Figure 20.3.

Support for the cassette interface is part of ROM BIOS and is accessed through software interrupt $15. Service 0 turns the motor on, and service 1 turns the motor off. No other parameters need be passed, and none are returned.

The two procedures for turning the relay on and off are essentially trivial invocations of software interrupt $15.

Figure 20.3

The PC's Cassette Port Interface



```
{->>>>RelayOn<<<<---------------------------------------------}
{                                                             }
{ Filename: RELAYON.SRC -- Last modified 7/14/88              }
{                                                             }
{ This one quite simply energizes the cassette relay, which   }
{ closes the contacts across pins 1 and 3 of the cassette DIN }
{ connector on the original PC.  (XT's and AT's do not have   }
{ the cassette relay.)  The specs allow for 1 amp through the }
{ relay, but having inspected the relay I would play it safe  }
{ and attempt no more than half an amp.                       }
{                                                             }
{                                                             }
{                                                             }
{-------------------------------------------------------------}

PROCEDURE RelayOn;

VAR
  Regs : Registers;

BEGIN
  Regs.AH := 0;          { AH = 0 -- Service that turns motor on }
  Intr($15,Regs)
END;
```

```
{->>>>RelayOff<<<<--------------------------------------------}
{                                                             }
{ Filename: RELAYOFF.SRC -- Last modified 7/14/88             }
{                                                             }
{ All we do here is de-energize the cassette relay, opening   }
{ the contacts across pins 1 and 3 of the cassette DIN        }
{ connector on the original PC.  (XT's and AT's do not have   }
{ the cassette relay.)                                        }
{                                                             }
{                                                             }
{                                                             }
{-------------------------------------------------------------}
```

```

PROCEDURE RelayOff;

VAR
  Regs : Registers;

BEGIN
  Regs.AH := 1;           { AH=1 -- Service that turns motor off }
  Intr($15,Regs)
END;
```

The only time I have used the cassette relay was in keying a small FM amateur radio transceiver, and in the process I learned that the PC is also a radio transmitter of considerable power. The square waves coursing about inside the PC are rich in harmonics, and unless you bypass the relay control lines at the DIN connector, you will hear the buzzsaw rasp of the timer interrupts in your receiver to the exclusion of just about everything else.

And although I had no trouble working at 3 watts RF, more powerful transmitters could well send enough loose RF energy into the PC to disrupt system memory. If things act strangely when you connect a transmitter to the cassette port, bypass; then bypass some more. Of course, keep in mind that the PC radiates RF through the air as well as though connecting cables. If your antenna is in the same room as the PC, you will almost certainly hear nothing but the PC regardless of how well the cassette port is bypassed. "Rubber duckie" antennas won't cut it!

20.5: MAKING DOS CALLS

Like most everything else in an 8088 environment, PC DOS is called through a software interrupt. We could, in fact, call PC DOS using Turbo Pascal's **Intr** software interrupt routine, by setting up values in a register structure and calling interrupt $21.

Turbo Pascal provides a somewhat more readable DOS call facility:

```
MSDOS(Regs);
```

where **Regs** is a record variable of the **Registers** type we examined in the previous section. Interrupt 21 does not have to be specified, as it is always the same for DOS calls and the interrupt calling code is built into the implementation of **MSDOS**.

The various functions performed by DOS each has a number. To invoke a function, this number must be loaded into register AH. Most individual DOS services require that additional parameters be loaded into various other registers before making the actual DOS call. We'll be providing numerous examples of **MSDOS** in use a little later.

DOS Error Messages and the DOSError Variable

Things go wrong. When a DOS function cannot complete successfully, DOS takes its best guess at what the problem is and returns an appropriate code in register AX. I say best guess because unusual situations have been known to knock DOS for a loop and return completely inappropriate error codes. You must therefore interpret DOS error codes carefully, and make provision for situations when the returned error codes don't bear any relation at all to the problem at hand.

Beginning with DOS 2.0, a set of 19 error code values was defined and used in a standard fashion across all the new DOS function calls first included in DOS 2.0. This set begins with function call $2F, Set DTA Address. Function calls with numbers lower than $2F probably *don't* return error codes from this set. By and large, the routines in unit **DOS** use the newer DOS function calls. If you use function calls with numbers prior to $2F, double check the DOS documentation to be sure you understand the errors such a call may return.

Table 20.2 summarizes the standard DOS error codes.

In summary: To make a DOS call, have defined a register variable of type **Registers** as described above, load the DOS service number into AH, load any other required parameters into the proper registers, and pass the structure to Turbo Pascal's **MSDOS** procedure:

Table 20.2
Standard DOS Error Codes

Code	*Meaning*
$00	No error; function call completed correctly
$01	Invalid function number
$02	File not found
$03	Path not found
$04	No handle available; all are in use
$05	Access denied
$06	Invalid handle
$07	Memory control blocks destroyed
$08	Insufficient memory
$09	Invalid memory block address
$0A	Invalid DOS environment
$0B	Invalid format
$0C	Invalid access code
$0D	Invalid data
$0E	<not used by DOS>
$0F	Invalid drive specification
$10	Attempt to remove current directory
$11	Not same device
$12	No more files to be found

```
Regs.AH := $36;  { "Get free disk space" service }
Regs.DL := $01;  { Request A: (drive 1) }

MSDOS(Registers);      { Make the DOS call }
```

If an error occurs, the error code will be returned in **AX**. Turbo Pascal's **DOS** unit contains a variable, **DOSError**, into which this error code is moved after a call to a DOS function that can return an error code.

IBM's documentation of DOS services is not always the best, especially if you're just learning your way around the PC at the system level. I don't have room to go over every single DOS service (there are about 85 in all) in this book. Experiment, but keep in mind that this is strong stuff. If you misunderstand the purpose of a parameter or put it in the wrong register, you could very easily blow your DOS session away and force a warm or cold reboot. Be careful, read it twice, and keep your cool!

Disk Size and Disk Free Space

DOS function call $36 does double duty in that it returns both the free space on a disk and the total possible space on a disk. The Turbo Pascal **DOS** unit contains two functions that use this function call to provide the same service at a higher level: **DiskFree** and **DiskSize**.

The **DiskSize** function returns the total number of bytes that may be stored on a specified disk:

```
FUNCTION DiskSize(Drive : Byte) : LongInt;
```

Parameter **Drive** is used to specify which drive in the system is to be sampled. A disk must be present in the drive for this function to work correctly as the disk will be accessed. In a somewhat contrary fashion, **Drive** does *not* contain the letter specifier for a disk drive, but a number indicating the drive, where 1 stands for drive A:, 2 stands for drive B:, and so on. To determine the size of the default drive, pass a 0 in the **Drive** function.

If the number passed in **Drive** does not correspond to a valid disk drive (for example, passing a 4 on a system with only drives A:, B:, and C:) **DiskSize** returns a −1.

DiskFree returns the number of free bytes available on a specified disk:

```
FUNCTION DiskFree(Drive : Byte) : LongInt;
```

The **Drive** parameter works in exactly the same way as it works in function **DiskSize**: Pass 1 to return the free bytes on drive A:, 2 for drive B:, and so on. A −1 value returned from the function indicates that the drive specified in **Drive** is invalid.

20.6: TIMES, DATES, STAMPS, AND FILES

"I can give you anything but Time," said Napoleon Bonaparte. Your PC won't provide you with an empire, but time . . . well, time is easy (the day of the week is another matter, but we'll get to that). Setting and reading the time and date from the PC's real-time clock are relatively simple matters given the machinery in the **DOS** unit. Using the clock's time values to measure duration (the distance in time from one point to another) is fairly simple and can be quite useful. This section will cover the details of dealing with time on the PC, with a little help from Turbo Pascal.

How the PC Keeps Time

Understanding *how* the PC keeps time is best begun by knowing *where* it "keeps" time. At $0040 : $006C is a 4-byte storage area that keeps a count of clock "ticks." These ticks occur roughly 18.2 times per second. Four bytes of zeroes at this location indicates midnight, and the count increases by 18.2 for every second past midnight. When DOS sets the clock to a particular time of day, it works backwards from the requested time to the number of ticks that should have occurred by that time each day and forces that number into the count storage location. When DOS needs to read the current time of day, it reads the four bytes from that storage location and converts the number of ticks to hours, minutes, and seconds. The date is kept elsewhere, as I'll describe shortly.

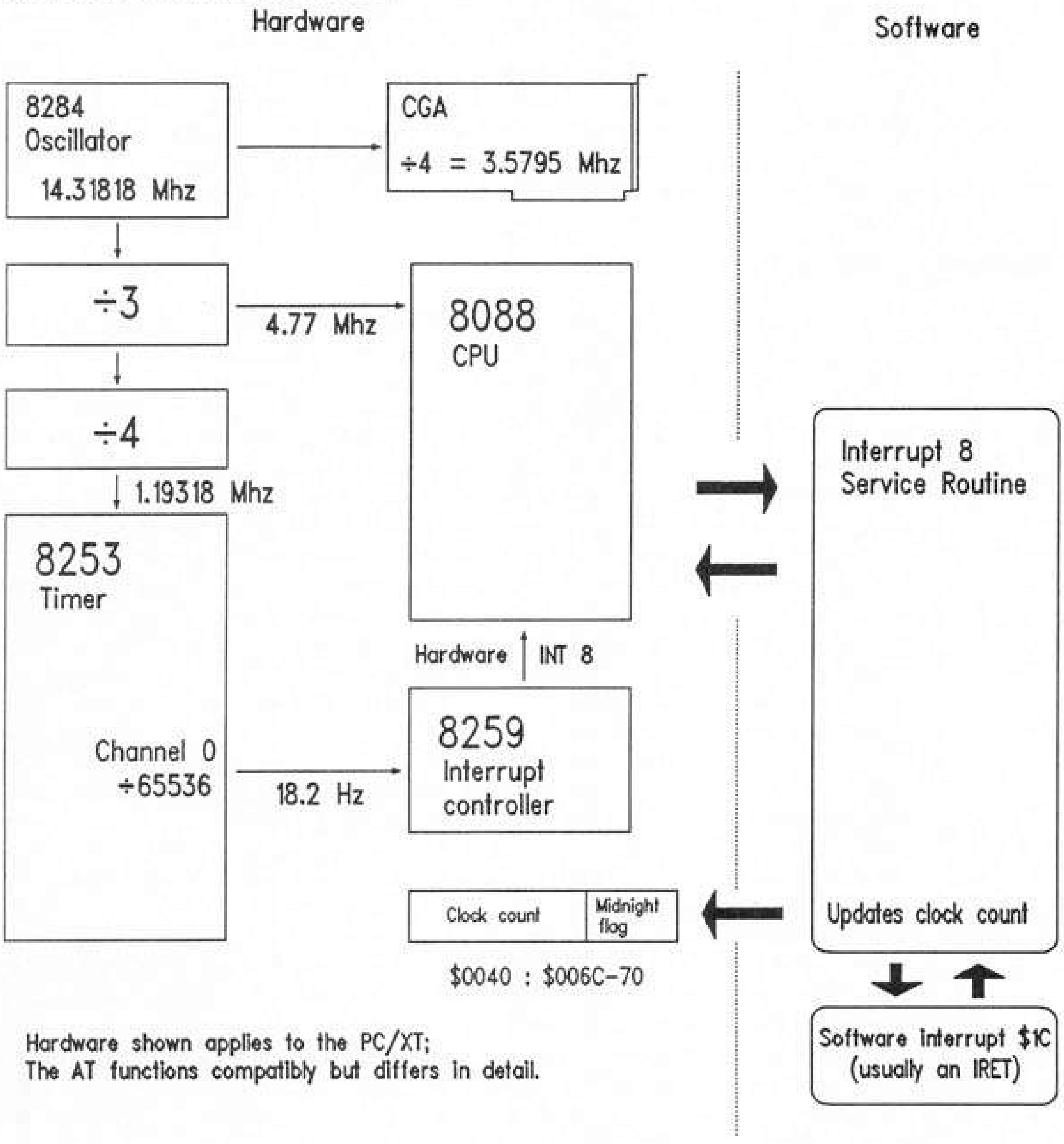
Now 18.2 ticks per second is a peculiar number, but it can be understood in the context of the PC's hardware. As Figure 20.4 indicates, the PC's timing mechanism provides several different frequencies for several different purposes, all from the same master reference frequency. On the PC motherboard is an 8284 clock controller chip with a crystal-controlled reference oscillator that produces a 14.31818 Mhz square wave signal. This frequency was chosen as a convenience to the Color Graphics Adapter, which requires a 3.579545 Mhz input to correctly create what is called the *color burst* signal. The color burst signal helps format color information onto the composite video output produced in the CGA. As some quick work with your Sidekick calculator will show, 3.579545 is 14.31818 divided by 4. Those unfamiliar with computer hardware should understand that dividing a frequency by a factor of 2 or 3 or some power of 2 or 3 is easily done with a couple of standard flip-flop circuits.

Another familiar number will appear if you divide 14.31818 by 3: 4.7727, which in Mhz is the CPU clock speed of the IBM PC. If you have ever wondered why the PC is so slow, allow yourself to get even angrier by answering yourself that it is due to a courtesy to a video format (composite color) that virtually no one uses anymore.

Well, one can accuse IBM of many things, but rarely are they guilty of foresight. It's done, so let's explore further. Dividing 14.31818 Mhz by 12 yields 1.19318 Mhz. This frequency is used as an input to a three-channel counter/timer chip called the 8253 (8254 on the PC/AT). The 8253 is a busy creature, as it controls the timer tick interrupt, dynamic RAM refresh, and speaker output all at once. Channel 0 is the one that controls the timer tick, which is of the most immediate interest.

Figure 20.4

The PC/XT Clock/Timer Mechanism


Hardware
Software
8284 Oscillator 14.31818 Mhz
CGA ÷4 = 3.5795 Mhz
÷3
4.77 Mhz
8088 CPU
÷4
1.19318 Mhz
8253 Timer
Channel 0 ÷65536
18.2 Hz
Hardware INT 8
8259 Interrupt controller
Interrupt 8 Service Routine
Updates clock count
Clock count
Midnight flag
$0040 : $006C-70
Software Interrupt $1C (usually an IRET)
Hardware shown applies to the PC/XT;
The AT functions compatibly but differs in detail.

Channel 0 of the 8253 divides the incoming signal (in this case 1.19318 Mhz) by up to 16 bits, or 65,536. Run that through your calculator and you'll see that it yields 18.206482. This is why we say that the PC clock ticks at "roughly" 18.2 timer ticks per second. The precise figure is given by the ratio 1,193,180 / 65,536. This 18.2 is the fewest output pulses per second obtainable from the 8253, given an input frequency of 1.19318 Mhz.

Therefore, the output of channel 0 of the 8253 is a series of pulses coming at roughly 18.2 per second. Each time a pulse is output by the 8253, the 8259 interrupt controller chip generates a hardware interrupt. The interrupt service routine for this interrupt (interrupt 8) is in the PC ROM BIOS. This interrupt service routine increments the clock reference count at $0040 : $006C.

With each tick, the INT 8 service routine tests the reference count for a particular value: 1,573,040 (in hex, $1800B0.) This is the number of timer ticks occurring in 24 hours 18.206482 times 86400, the number of seconds in a 24-hour day. When it detects this value, the count "rolls over" to zero, on the assumption that midnight has just occurred. It also writes a 1 into another location in memory at $0040 : $0070, immediately above the clock reference count. This is the "midnight flag" that indicates that midnight has occurred since the last time the clock was set.

The INT 8 service routine also generates a software interrupt $1C with each clock tick. This interrupt ordinarily does nothing, and the interrupt vector associated with it initially points to an IRET instruction, which does nothing but bounce control back to INT 8 like a ping-pong ball. However, it is possible to install a service routine for INT $1C that will execute some task 18.2 times per second. By making this task a process scheduler, some minimal multitasking support is possible. Of course, as anyone who has ever tried to run Unix on a 4.77 Mhz IBM PC should be aware, you can't expect much in terms of performance. The interrupt can be very useful in certain limited applications, like print spoolers. "Background" tunes can also be played by using interrupt $1C to feed new timer values to 8253 channel 2, which controls the speaker. You can consider that application a "tone spooler."

Another note: Some people have leapt to the conclusion that 65,536 clock ticks occur in an hour, which is close but not true, since the true figure is 65,543 clocks and change.

Obviously, the clock count is only maintained while the PC is powered-up and working properly. During periods when the power is off, the correct time must be maintained elsewhere. This elsewhere is usually on a multifunction board, although some PC clone motherboards include a battery-operated clock chip. A small lithium battery keeps an oscillator and a clock chip alive, and when the PC is powered up, a transient utility or a DOS device driver reads the battery-powered clock and loads its value into the clock reference count at $0040:$006C.

Reading and Setting the Time through DOS

Turbo Pascal's **DOS** unit provides several functions connected with the PC's time and date: **GetDate**, **SetDate**, **GetTime**, and **SetTime**. Internally, these amount to very little more than "wrappers" around four DOS function calls made through the **MSDOS** procedure. They exist to hide the details of loading and unloading values between the register structure and more tractable parameters, but you should be aware of what is going on beneath the surface.

GetTime returns the time as currently maintained in the PC by using DOS Function **44** ($2C). It is declared this way:

```
PROCEDURE GetTime(VAR Hour,Minute,Second,Sec100 : Word);
```

Its information ultimately comes from the clock reference count at $0040 : $006C, but DOS converts the raw count of ticks since midnight into the familiar hours, minutes, seconds, and a slightly ersatz hundredths of seconds figure.

The hundredths figure is questionable because the PC's clock doesn't really resolve to a single hundredth of a second. Since there are only 18.2 clock ticks per second, the duration of a single tick is about 0.055 seconds (the reciprocal of 18.2). What DOS does is interpolate a given tick to its closest decimal equivalent within a second. This figure can be useful as long as you understand that there is that inescapable roundoff to the closest near-twentieth of a second.

Retrieving the time through DOS is trivial, then—as long as all we need is time in the form of four numbers. In everyday programming, time values are useful in a number of forms. A time-fetch routine can start with DOS's basic numeric time values and calculate these additional forms.

One common use of time is simply to display it, in a convenient format, for a human reader. The format I like is the one most familiar to IBM PC users: The DOS DIR time format, which is a colon-separated 12-hour format with a single-letter AM/PM indicator: **12:17p** Creating a string containing such a time format can be done with a little use of Turbo's **Str** built-in procedure and some concatenation.

There is one more form in which time can be expressed that you may find useful: a "time stamp" that expresses the time in 16 bits so that one such stamp may be compared with another to see which is later in time. This is a little tricky, as it isn't quite possible to express all four time units (hours, minutes, seconds, and hundredths of seconds) in one 16-bit time stamp. The matter of time stamps is important in dealing with DOS files, and I will devote considerable time to it later in this section.

In my own programming, I have combined these three different expressions of time into a single Pascal record:

```
TimeRec = RECORD
            TimeComp   : Word;  { DOS time stamp format }
            TimeString : String80;
            Hours,Minutes,Seconds,Hundredths : Integer
          END;
```

Filling such a record with the current time is done in two steps. The first step is simply to use Turbo Pascal's **GetTime** function to fill in the hours, minutes, seconds, and hundredths of seconds:

```
VAR
  TimeNow : TimeRec;

WITH TimeNow DO GetTime(Hours,Minutes,Seconds,Hundredths);
```

GetTime itself only makes the DOS call and loads the **TimeRec** integer fields. A separate routine, **CalcTime** is used to calculate the string version of the time value and the time stamp value. I broke this out as a separate procedure so that time values originating elsewhere than the system clock can be put into the same format as system time values.

```
{->>>>CalcTime<<<<------------------------------------------}
{                                                          }
{ Filename: CALCTIME.SRC -- Last Modified 7/7/88           }
{                                                          }
{ This routine "fills out" a TimeRec passed to it with only  }
{ the DOS time values (hours, minutes, seconds, hundredths)  }
{ valid.  It generates the TimeComp and TimeString fields.   }
{                                                          }
{     TimeRec = RECORD                                     }
{                 TimeComp   : Word;      (DTA time stamp)  }
{                 TimeString : String80;                   }
{                 Hours,Minutes,Seconds,Hundredths : Integer }
{               END;                                       }
{                                                          }
{ which, of course, also requires definition of type String80. }
{                                                          }
{                                                          }
{                                                          }
{----------------------------------------------------------}

PROCEDURE CalcTime(VAR ThisTime : TimeRec);

TYPE
  String5 = String[5];

VAR
  Temp1,Temp2 : String5;
  AMPM        : Char;
  I           : Integer;

BEGIN
  WITH ThisTime DO
    BEGIN
      I := Hours;
      IF Hours = 0 THEN I := 12;   { "0" hours = 12am }
      IF Hours > 12 THEN I := Hours - 12;
      IF Hours > 11 THEN AMPM := 'p' ELSE AMPM := 'a';
      Str(I:2,Temp1); Str(Minutes,Temp2);
      IF Length(Temp2) < 2 THEN Temp2 := '0' + Temp2;
      TimeString := Temp1 + ':' + Temp2 + AMPM;
      TimeComp :=
        (Hours SHL 11) OR (Minutes SHL 5) OR (Seconds SHR 1)
    END
END;
```

You should note that while the integer value **Hours** is left in 24-hour format, the string representation of the time converts its hours figure to a 12-hour format with the a and p to indicate AM or PM. **CalcTime** creates the time stamp by shifting the hours, minutes, and seconds numbers into their proper orientation and then ORing them into what amounts to a bitmap, as I'll explain in detail on page 379.

Setting the time is easily done with the **SetTime** procedure in the **DOS** unit:

```
PROCEDURE SetTime(Hour,Minute,Second,Sec100 : Word);
```

Keep in mind that while **GetTime** and **SetTime** are included in the **DOS** unit, you will have to predefine **TimeRec** somewhere. On the listings diskette for *Complete Turbo Pascal* I have provided a file called **TIMEREC.DEF** containing the required definitions:

```
TimeRec = RECORD
            TimeComp   : Word;       { DOS time stamp format }
            TimeString : String80;
            PM         : Boolean;
            Hours,Minutes,Seconds,Hundredths : Integer
          END;
```

Reading and Setting the Date through DOS

There's a little-understood but critical difference between the way the PC handles the time and the date. Actually, the difference is that the PC *doesn't* handle the date at all in the same sense that it keeps time. Time on the PC is very nearly a hardware function. The hardware timer generates time-of-day interrupts (INT 8), which are serviced by a routine in ROM, which updates the clock reference count at $0040:$006C. The only information relevant to date-keeping is a single flag at $0040:$0070. This is the *midnight flag*, and it is set by the time-of-day interrupt service routine to a value of 1 when the clock reference count rolls over to 0 at midnight.

That's it. The date, if it is to be kept at all, must be kept either by the operating system or by the application software. PC DOS maintains the current date, but it keeps the date value in a peculiar place: inside COMMAND.COM, the DOS command processor program. This location can be found by some DEBUG snooping, but it's different for each version of DOS and, like any "undocumented" DOS location, it could disappear entirely or reappear in a completely incompatible form without any warning. While you can muck around with the clock reference count and derive your own time values from it by going directly to memory at $0040:$006C, the *only* way to set or read the date is to do it DOS's way, through DOS function calls 42 ($2A) and 43 ($2B).

DOS uses the midnight flag to determine when to increment its date value. If DOS goes out to read the clock reference count through ROM BIOS service $1A and discovers that the midnight flag has been set, it increments its date value within COMMAND.COM. It certainly does this at boot time, but for a PC that is left on continuously, it is unclear when DOS actually goes out and looks at the clock reference count.

There is an excellent reason not to use ROM BIOS service $1A to read the clock reference count in your own programs. *Any* reading of the count through this service, by you or by DOS, clears the midnight flag to 0. If your program uses BIOS service $1A before DOS gets around to updating its date value (again, on a PC that stays on all the time), your date will fall one day behind. For this reason, consider BIOS service $1A DOS's private creature: If you must read the clock reference count, go directly to memory with **MemW** at $0040:$006C.

When you read the date from DOS through DOS service $2A, it provides the year, month, day, and day-of-the-week, all as integer quantities. The mechanism is a procedure called **GetDate** in the **DOS** unit:

```
PROCEDURE GetDate(VAR Year,Month,Day,DayOfWeek : Word);
```

As with the time, I created a record type to hold DOS's date values, two string representations of the date, and a "date stamp" word (described a little later) that follows the DOS date stamp format used in its directory entries:

```
TYPE
  DateRec = RECORD
              DateComp       : Word;
              LongDateString : String80;
              DateString     : String80;
              Year,Month,Day : Integer;
              DayOfWeek      : Integer
            END;
```

As with my **TimeRec**, filling a **DateRec** with information from DOS is a two-step process: First, **GetDate** fills the integer fields of a **DateRec** with the correct values:

```
VAR
  Today : DateRec;

WITH Today DO GetDate(Year,Month,Day,DayOfWeek);
```

A separate routine, **CalcDate**, calculates the date stamp and the two string representations of the date:

```
{->>>>CalcDate<<<<--------------------------------------------}
{                                                              }
{ Filename: CALCDATE.SRC -- Last Modified 7/11/88              }
{                                                              }
{ This routine fills in the DateString, LongDateString, and    }
{ DateComp fields of the DateRec record passed to it.  It      }
{ requires that the Year, Month, and Day fields be valid on    }
{ entry.  It also requires prior definition of types DateRec   }
{ and String80.  DateString is formatted this way:             }
{                                                              }
{     Wednesday, July 17, 1986                                 }
{                                                              }
{ DateRec is declared this way:                                }
{                                                              }
{     DateRec = RECORD                                         }
{                 DateComp       : Word;                       }
{                 LongDateString : String80;                   }
{                 DateString     : String80;                   }
{                 Year,Month,Day : Integer;                    }
{                 DayOfWeek      : Integer                     }
{               END;                                           }
{                                                              }
{ DayOfWeek is a code from 0-6, with 0 = Sunday.               }
{ DateComp is a cardinal generated by the formula:             }
{                                                              }
{     DateComp = (Year-1980)*512 + (Month*32) + Day            }
{                                                              }
{ It is used for comparing two dates to determine which is    }
{ earlier, and is the same format used in the date stamp in    }
{ DOS directory entries.                                       }
{                                                              }
{                                                              }
{                                                              }
{--------------------------------------------------------------}

PROCEDURE CalcDate(VAR ThisDate : DateRec);

TYPE
  String9 = String[9];

CONST
  MonthTags : ARRAY [1..12] of String9 =
    ('January','February','March','April','May','June','July',
     'August','September','October','November','December');
  DayTags   : ARRAY [0..6] OF String9 =
    ('Sunday','Monday','Tuesday','Wednesday',
     'Thursday','Friday','Saturday');

VAR
  Temp1 : String80;

BEGIN
  WITH ThisDate DO
    BEGIN
      DayOfWeek := DateToDayOfWeek(Year,Month,Day);
      Str(Month,DateString);
      Str(Day,Temp1);
      DateString := DateString + '/' + Temp1;
      LongDateString := DayTags[DayOfWeek] + ', ';
      LongDateString := LongDateString +
        MonthTags[Month] + ' ' + Temp1 + ', ';
      Str(Year,Temp1);
      LongDateString := LongDateString + Temp1;
      DateString := DateString + '/' + Copy(Temp1,3,2);
      DateComp := (Year - 1980) * 512 + (Month * 32) + Day
    END
END;
```

The two string representations cover the two most common expressions of a date in human-readable form: A "short" form with slashes: **6/29/87**; and a long form spelling it all out: **Friday, January 2, 1987**. Subsets of the long form can easily be extracted (for example, without the day-of-the-week or simply the month and the day) by using the Turbo Pascal built-in string function **Copy**.

As with **TimeRec**, I have added a small include file to the listings diskette for this book, containing the **DateRec** declaration:

```
DateRec = RECORD
              DateComp        : Word;
              LongDateString : String80;
              DateString      : String80;
              Year,Month,Day : Integer;
              DayOfWeek       : Integer
            END;
```

Time and Date Stamps

A more intriguing use of time is in the comparison of two time values to see which is the older of the two. Using the four different integer values in a comparison would involve a lot of IF/THEN testing, since if the hours are equal, then the minutes have to be tested, and if the minutes are also equal, then the seconds have to be tested, and so on. It would be handy if there were a way to combine the elements of a time value into a single number that could be used in a single compare operation to tell if one value is older than another. A number like this is often called a *time stamp.*

The obvious way is simply to express a time as the number of hundredths of a second in all the hours, minutes, and seconds of a day. In a 24-hour day this works out to 8,640,000. This would fit nicely in a 32-bit long integer. However, having a time stamp that fits in 16 bits would be faster to manipulate and more compact.

We can drop the hundredths, as they're a little too ersatz for my tastes anyway. This cuts the figure to 86,400 which is agonizingly close to 65,536 but still too high to express in 16 bits.

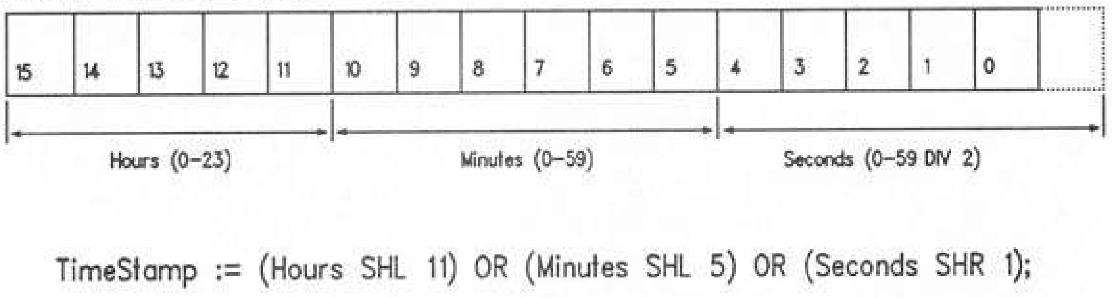
Compromises must then be made. By dividing the maximum number of seconds in the time stamp by two (to 30 instead of 60), the number of bits required to express the seconds in a minute drops from 5 to 4. Granting this slight reduction in the time stamp's precision, we can now express the number of hours in day in 5 bits (0 through 23); the number of minutes in an hour in 6 bits (0 through 59); and the number of "seconds" in 5 bits (0 through 29), such that 5+6+5 = 16 (see Figure 20.5).

Losing hundredths and every other second means that if two time stamps are within a second of one another, they will be identical. This might not be the time stamp method of choice in a system making rapid laboratory measurements in real-time, but for expressing the creation time of a file (which usually takes more than one second to create, write, and close anyway), it works quite well.

A 16-bit date stamp can be constructed by shifting the years figure 9 bits to the left (actually by multiplying it by 512, which is the same thing); adding that to the

Figure 20.5

The DOS Time Stamp Format



months figure shifted left by 7 bits (i.e., multiplying the months figure by 64); and then adding in the days value unchanged. Because the "19" portion of the year value is not incorporated into the date stamp, there are plenty of bits to go around.

I can't take credit for the 16-bit time stamp and date stamp formats described here. They are used by DOS to stamp its own files. And although I described them separately for clarity's sake, DOS actually stores a file's time and the date stamp side-by-side so that they can be used together to determine which of two files was last updated more recently. Turbo Pascal 4.0 and later versions have a 32-bit long integer type, which can hold DOS's time and date stamps quite handily. Two such combined stamps stored in long integers can be compared with a single relational operator, rather than having to first compare the date stamps and then the time stamps.

Turbo Pascal's support for DOS file time and date stamps consists of four procedures, all in the **DOS** unit: **GetFTime**, **SetFTime**, **PackTime**, and **UnpackTime**.

A file's time and date stamps are read together into a single long integer parameter by using the **GetFTime** procedure in the **DOS** unit:

```
PROCEDURE GetFTime(VAR F; Time : LongInt);
```

The untyped parameter **F** must be an *opened* file of any type: text, typed, or untyped. Its time and date stamps are read into long integer parameter **Time**, with the date stamp occupying the high-order 16 bits.

The long integer stamps retrieved from files can be compared directly:

```
VAR
  FileX, FileZ : Text;
  FileXStamp, FileZStamp : LongInt;

Assign(FileX,'FILEX.TXT'); Reset(FileX);
Assign(FileZ,'FILEZ.TXT'); Reset(FileZ);
```

```
GetFTime(FileX,FileXStamp);
GetFTime(FileZ,FileZStamp);
IF FileXStamp > FileZStamp THEN
  Writeln('FileX is older than FileZ.') ELSE
  Writeln('FileZ is older than FileX.');
```

Of course, you may wish to treat the time and date portions of the stamp separately, and you will almost certainly want to separate the stamp into integer quantities for year, month, day, hours, minutes, seconds, and so on. Turbo Pascal provides two translation routines for combining the separate time and date values into a long integer stamp, and separating them out again.

The **PackTime** procedure creates a long integer stamp from separate time and date values:

```
PROCEDURE PackTime(VAR DTData : DateTime; VAR Stamp : LongInt);
```

The **DateTime** type is a record defined in the interface section of the **DOS** unit:

```
DateTime = RECORD
             Year,Month,Day,Hour,Min,Sec : Word
           END;
```

PackTime takes values from a **DateTime** record and combines them into long integer parameter **Stamp**.

Going the other way involves procedure **UnpackTime**:

```
PROCEDURE UnpackTime(Stamp : LongInt; VAR DTData : DateTime);
```

The parameters are the same as those used with **PackTime**. Here, however, the input parameter is the long integer **Stamp** and the output is returned in **DTData**.

Just as the time and date stamps of a file may be returned to your Turbo Pascal program by **GetFTime**, there is a **DOS** unit function to set the stamps to some specified value:

```
PROCEDURE SetFTime(VAR F; Time : LongInt);
```

This procedure is very much the reverse of **GetFTime**, in that the parameters are the same types and have the same meaning. Only the direction the data is moving changes.

A "Touch" Utility

One good example of several of these time and date routines in use lies in a *touch* utility. On the surface, **Toucher** is very simple: It opens a file specified on the command line, reads the current time and date, and then sets the time and date stamps of the opened file to the current time and date read from the system clock. If, for example, you used **Toucher** on a file that was last modified sometime last May, any further DIR listings of that file would seem to indicate that it had been modified the moment **Toucher** "touched" it.

We define a **DateTime** record (the definition is in the **DOS** unit) named **Now**, and then use the two procedures **GetDate** and **GetTime** to fill in the time and date values for right now (i.e., when **Toucher** executes). The time and date stamp values are then "packed" into a single long integer value using the **PackTime** procedure. Finally, **SetFTime** applies the combined long integer stamp to the file itself. Close the file, and **Toucher** has done its work.

Now, why all this fuss? Of what possible use is a utility that simply tells a lie about the last time a file was modified?

Well, a lie is what it is, but a lie with a purpose. As I will be describing in Part 3, Turbo Pascal 4.0 and 5.0 contain a "make" facility that helps you handle the complexity of a large project with many interdependent source and object files. The Make logic within the compiler chooses to compile or not compile a given file depending on whether any files on which it is dependent were modified *after* the file in question. In other words, if module A USES module B, and module B was modified more recently than Module A, then the compiler will choose to compile both modules; otherwise, it will only recompile module A.

However, suppose you would like to compile both? Apart from manually invoking the compiler separately, which bypasses the Make logic altogether, you can artificially force module B's source file to look like it was modified more recently than module A, causing Make to invoke the compiler first for module B and then for module A. The way, of course, is to *touch* module B with **Toucher**.

```
{--------------------------------------------------------------}
{                          TOUCHER                             }
{                                                              }
{          'Touch' utility for DOS unit demonstration          }
{                                                              }
{                          by Jeff Duntemann                   }
{                          Turbo Pascal V4.0                   }
{                          Last update 7/1/88                  }
{                                                              }
{                                                              }
{                                                              }
{--------------------------------------------------------------}


PROGRAM Toucher;

USES DOS;

VAR
  I      : Integer;
  Stamp  : LongInt;
```

```
   Now    : DateTime;
   Target : File;
   Sec100 : Word;
   DayOfWeek  : Word;

 BEGIN
   IF ParamCount < 1 THEN
     BEGIN
        Writeln('>>TOUCHER  V1.00 by Jeff Duntemann');
        Writeln('           From the book COMPLETE TURBO PASCAL 5.0');
        Writeln('           Scott, Foresman & Company, 1988');
        Writeln('           ISBN 0-673-38355-5');
        Writeln;
        Writeln('  Calling Syntax:');
        Writeln('  TOUCHER <filename>');
        Writeln;
        Writeln('  TOUCHER is a "touch" utility that replaces ');
        Writeln('  the time/date stamp of a file with the current');
        Writeln('  date and time.  This can be used to force a remake');
        Writeln('  on a project depending on that file.');
     END
   ELSE
     BEGIN
       Assign(Target,ParamStr(1));
       {$I-} Reset(Target); {$I+}
       I := IOResult;
       IF I <> 0 THEN
         BEGIN
           Writeln('>>Error!  Named file cannot be opened...');
         END
       ELSE
         BEGIN
           WITH Now DO GetTime(Hour,Min,Sec,Sec100);
           WITH Now DO GetDate(Year,Month,Day,DayOfWeek);
           PackTime(Now,Stamp);
           SetFTime(Target,Stamp);
           Close(Target);
         END
     END
END.
```

Calculating the Day of the Week

DOS contains a slightly flawed algorithm for calculating the day-of-the-week given the date, so that when you read the date through **DOS** unit procedure **GetDate**, which uses DOS function $2A, you get a day-of-the-week indicator along free. Sunday is represented as a 0, and Saturday as a 6. Under the assumption that I would at some point need to fill a **DateRec** that did not come from DOS function $2A, I needed a means to calculate the day-of-the-week from the date.

There are a number of published methods of doing this, all of them messy and involved. Peter Norton was kind enough to point out that DOS already knows how to figure the day-of-the-week. All you have to do is save the current date, set the DOS date to the date in question, read it back with DOS's calculated day-of-the-week code, and then reset the DOS date to the true date.

This is quick and convenient, but there are two catches: 1) DOS does not "understand" dates prior to 1/1/80, which is "Year Zero" as far as the PC is concerned; and 2) the DOS date algorithm does not return the correct day-of-the-week for leap year day, February 29.

The first problem could be important, if you need to deal with dates prior to 1980, say in connection with birth dates, old contract dates, and so on.

The second problem is minor, since DOS is entirely consistent about its errant ways: The day-of-the-week returned for leap year day is always exactly one day code higher than it should be. In other words, if DOS returns a 0, the real code is 6; if DOS returns a 1, the real code is zero, etc. This correction is simplified by the fact that the year 2000 is special. Ordinarily, "century years" are *not* leap years, even though they are divisible by 4, since a day has to be dropped every 100 years to make things come out even. But every 400 years, a century year *is* a leap year. (Again, to compensate for a very *very* minor cumulative error in the calendar.) 1600 was the last time this happened, and 2000 is next. For this reason, 2000 may be treated as an ordinary leap year.

I trust you will not be programming in Turbo Pascal in the year 2100.

```
{->>>>DateToDayOfWeek<<<<------------------------------------}
{                                                            }
{ Filename : DAYOWEEK.SRC -- Last Modified 7/11/88           }
{                                                            }
{ This function "calculates" the day of week from the month, }
{ day, and year values passed to it.  The actual calculation }
{ is done by DOS, by setting the current date in the PC to the }
{ date passed, and then reading back the current date        }
{ to get the day of week in AL.  (The real current date was  }
{ read and saved and is restored before control returns to the }
{ caller.)  The bulk of the routine deals with the fact that }
{ DOS cannot correctly calculate the day of the week for any }
{ leap year day.  Fortunately, it's consistent in its error, }
{ and the error can be easily corrected for.                 }
{                                                            }
{                                                            }
{                                                            }
{------------------------------------------------------------}

FUNCTION DateToDayOfWeek(Year,Month,Day : Integer) : Integer;


VAR
  SaveDate,WorkDate : Registers;
  DayNumber         : Integer;
  LeapYearDay       : Boolean;

CONST
  DayArray : ARRAY[1..12] OF Integer =
    (31,28,31,30,31,30,31,31,30,31,30,31);
```

```

BEGIN
  LeapYearDay := False;
  IF (Month = 2) AND ((Year MOD 4)=0) AND (Day = 29) THEN
    LeapYearDay := True;
  IF (NOT LeapYearDay) AND (Day > DayArray[Month]) THEN
    DateToDayOfWeek := -1
    ELSE
      BEGIN
        WorkDate.AH := $2B;
        SaveDate.AH := $2A;  { Saves date encoded in registers }
        MSDOS(SaveDate);             { Fetch & save today's date }
        WITH WorkDate DO
          BEGIN
            CX := Year;       { Set the clock to the input date }
            DH := Month;
            DL := Day;
            MSDOS(WorkDate);
            AH := $2A;        { Turn around and read it back }
            MSDOS(WorkDate); {  to find the day-of-week indicator }
            DayNumber := AL; {  in AL. }
            IF LeapYearDay THEN     { Correct for DOS's leap year bug }
              IF DayNumber = 0 THEN DayNumber := 6
                ELSE DayNumber := Pred(DayNumber);
            DateToDayOfWeek := DayNumber
          END;
        SaveDate.AH := $2B;  { Restore clock to today's date }
        MSDOS(SaveDate);
      END
END;
```

20.7: UNDERSTANDING DOS DISK DIRECTORIES

There are two ways to look at disk files, the logical and the physical. The physical file is a collection of magnetic disturbances arranged on a floppy disk. The logical file is a sequence of sectors stored on a disk, where a sector is a "slice" of a disk file comprising some number of bytes of data. The physical location of any individual sector on the disk itself is of no concern in the logical view of the file. The physical file has no name; it is only a collection of sectors scattered like buckshot across the face of the disk.

DOS intermediates between the two views of a file. It is possible to go around DOS and take charge of the physical reality of the file's naked sectors, but there's plenty of danger and very little profit in that, unless you intend to play copy protection games; and I think I'd rather run a school for child molestors than teach people how to copy protect their diskettes.

As a result, I won't be going into the physical view of a file here, nor explain the FAT, which is the ultimate arbiter between physical and logical, nor most of the other DOS internals that could fill a book on their own. At the logical level, the most important entities aside from the disk files themselves are disk directories. By and large, this section concerns disk directories, how DOS formats them, and how the information contained there can be used.

Root Directories and Subdirectories

A directory is like a phone book; there is one entry in a directory on a disk for every file on that disk, and this entry contains information essential for opening and using that file. There are two types of directories available under DOS. Every disk volume (which is a mass storage entity responding to a drive specifier like A:, B:, C:, etc.) has one and only one root directory. The root directory is of a fixed size, although the size of root directories varies depending on the type of disk you're dealing with.

Subdirectories may exist within the root directories, or within other subdirectories. They are tools for organizing groups of disk files into logical structures, revealing those of interest while hiding those that are not needed for the time being. They also serve to keep unused directory space from hogging precious disk space. As with dynamic variables in Pascal, subdirectories can be created as needed in unlimited numbers (because they can be nested) and erased when no longer required.

A subdirectory is in fact a special type of file. Until DOS 3.0, in fact, subdirectories could be read and processed by the same function calls used to process ordinary data files. As part of DOS's ongoing evolution toward a protected-mode multitasking operating system, this and other shortcuts have been removed. DOS provides adequate tools for dealing with subdirectories at a Turbo Pascal program level. For disk diagnostic routines and other very low-level utilities, stronger measures (such as tracing through the labyrinth of the FAT) are required.

Directory Entries Versus DTA

Echoing the logical/physical dichotomy of the disk file, there are two faces to a directory entry. One is the entry as it is actually stored on the disk, and the other is the way DOS shows us that entry when we search for it and find it. There are two reasons for this split personality: 1) the "real" directory entry contains information (specifically, the starting FAT cluster) that DOS would prefer to keep to itself; and 2) there is information that is useful for repeated directory searches that is created and deleted after the searches are completed, and does not need to be stored away on disk.

The directory entry as it stored on disk is not of any serious concern except for the writing of low-level disk utilities like the Norton Utilities. For everyday work with files, the second manifestation of the directory entry, the entry as it is shown to us in a place called the Disk Transfer Area, is the one I will be describing and using here.

As you might expect, there are DOS function calls that go out to the directory and search for entries that match a particular filespec. These calls are embodied in a pair of procedures in Turbo Pascal's **DOS** unit (**FindFirst** and **Findnext**) that read and inspect disk directories. These two procedures allow a program to go out and see what is actually on the disk before attempting to open something, or to make lists of file names for the program to manipulate in some way. I'll be describing them in detail a little later in this section.

FindFirst and **FindNext** place information about files in a table called the Disk Transfer Area, or DTA. The DTA table is the closest DOS will let us get to a directory entry without goosing it beneath the belt. The DTA's organization is best explained and used by setting it up as a Pascal record:

```
SearchRec   = RECORD
                Fill : ARRAY[1..21] OF Byte;
                Attr : Byte;
                Time : LongInt;
                Size : LongInt;
                Name : String[12]
              END;
```

This record type is defined in the **DOS** unit, and may be used to declare variables of this type for use in programs incorporating the **FindFirst** and **FindNext** procedures.

The first 21 bytes are reserved, and DOS *means* that. Don't think that in this case "reserved" means "unused." DOS uses those 21 bytes to hold information over from one type of DOS call to the next, and if you try tucking something away in there, your file searches won't work.

Time should be familiar from the previous section. It is the combined time and date stamp DOS keeps for every file in its directory entry, and may be used in a numeric comparison to determine which of two files was modified more recently.

Size is the size of the file as it exists on disk, in bytes.

Name is the name of the file. This includes the dot *if* there are more than 8 characters in the name. The file name is an "ASCIIZ" string. This means that it is a string of characters without a length byte, plus a "null" character, character 0, appended to the end to indicate that it does in fact end somewhere.

Attr is the DOS file attribute byte. File attributes are "colors" that a file may take on under certain special conditions so that it may be treated in special ways. Six of the eight bits in the attribute byte are significant, and they are outlined in Figure 20.6. Bits are considered *active* if set to 1.

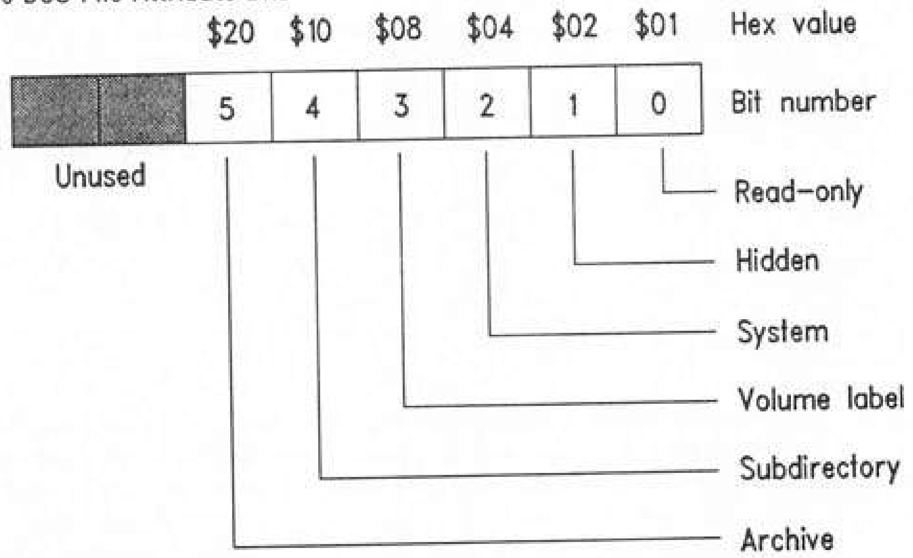
Bit 0, if set to 1, marks a file as *read-only,* meaning it cannot be deleted or modified via normal DOS operations. To delete or change the file you must first use a particular DOS call (CHMOD) to zero out the read-only bit, or use a DOS utility called ATTRIB from the command line to lower the read-only flag.

Bit 1 marks a file as *hidden,* which means the file exists but cannot be seen or modified by most DOS operations. Some DOS calls will detect hidden files (those functions calls used by **FindFirst** and **FindNext**, notably) but in most cases you need to use CHMOD to "unhide" them before you can do much with them. Out of sight, out of mind, I guess; and that old devil ERASE *.* won't touch hidden files.

Bit 2 marks a file as a *system* file, but has no real significance that I can identify. Peter Norton says it is a holdover from CP/M-80, but that's unclear to me. Ignore it.

Figure 20.6

The DOS File Attribute Bits



Bit 3, if set, marks the file as the current *volume label*, an 11-character name that can be given to any disk volume. Volume labels are little used, in part because of a DOS bug that prevents you from altering a volume label once one is created. More on this in Section 20.8.

Bit 4 marks a file as a *subdirectory*. Subdirectories are, in fact, files, and in DOS 2.X they could even be read as files using ordinary DOS calls. DOS 3.X will not allow subdirectories to be opened and read like files, but this attribute bit still allows you to identify such files as subdirectories, as we'll see in designing the **Locate** program shortly.

Bit 5 is the *archive* bit, and is set to 1 every time a file is changed somehow. The idea is that a backup utility can inspect a directory and only back up files with the archive bit set. Then, once a file is archived, the utility can set its archive bit back to 0, so that it will be ignored by the next archiving operation. Once the file is again modified, its archive bit will be set back to 1, making it fair game for the backup utility. Unless you use a backup utility, this bit will be set to 1 on virtually all files all the time.

Inspecting and Changing a File's Attribute Byte

The attribute byte says a lot about a file, and it is important enough so that Turbo Pascal's **DOS** unit contains two procedures for reading and setting the attribute byte. Reading the attribute byte from a file's disk directory entry is done with **GetFAttr**:

```
PROCEDURE GetFAttr(VAR F; VAR Attr : Word);
```

Here, **F** is an untyped **VAR** parameter (see Section 20.3) that may take any type of file variable, once the file variable has been assigned some physical file name and opened. When invoked, **GetFAttr** returns the attribute byte as the low-order byte of **Word** variable **Attr**. It is unclear why **Attr** is a **Word** and not a **Byte**.

Changing the attribute byte of a file is done with **SetFAttr:**

```
PROCEDURE SetFAttr(VAR F; Attr : Word);
```

Again, **F** can take any assigned and opened file variable, and **Attr** contains the desired attribute information in its low eight bits.

After using either **SetFAttr** or **GetFAttr**, you should check the **DOSError** variable (defined in the **DOS** unit) to make sure that the operation went through successfully. This is especially important since nothing prevents you from passing *anything* in variable **F**: integers, records, sets, characters, whatever. Obviously, only a file variable will do, but it is up to you to make sure a file variable is passed in **F**.

Telling DOS about the DTA

When a program begins running, DOS sets up a default DTA at offset $80 into the file. This is the same address at which the *command line tail* is placed, which is why you must pick up and save the command parameters *before* you begin searching for or opening and reading any files. While it is certainly possible to use the DTA at offset $80, I feel it is a better idea to declare a **SearchRec** as a variable, and then instruct DOS to use that variable as its DTA.

A DOS call, $1A, exists for this purpose. You can use it to set up a **SearchRec** as the current DTA:

```
VAR
  CurrentDTA : SearchRec;
  Regs       : Registers;

.   .   .   .   .

WITH Regs DO
  BEGIN
    AH := $1A;
    DS := Seg(CurrentDTA);
    DX := Ofs(CurrentDTA);
  END;
MSDOS(Regs);
```

The idea here is to place the address of the first byte of the new DTA in DS (the segment portion) and DX (the offset portion.) As with all DOS calls, the number of the function goes in register AH.

You might get a little nervous about placing a new value in DS, when DS holds the segment address for Turbo Pascal's data segment, but in fact the current value of DS is saved and restored by the code generated for the **MSDOS** procedure by Turbo Pascal.

After this code is executed, any time DOS requires the use of a DTA, it will use the one mapped out at the address passed in DS : DX; in this case, record variable **CurrentDTA**. This particular **SearchRec** was declared as a variable, but you can also pass DOS the addresses of **SearchRecs** passed as parameters or declared as record constants.

A Better Record Format for Directory Information

The DTA format embodied in **SearchRec** is what DOS gives us, but it's not necessarily the best that can be done. The combined time and date stamp is very compact, but not very useful except for determining the older of two files. To be used at all it must be processed by the **UnpackTime** procedure in the **DOS** unit, and even then the values are strictly numeric, when what is often needed is a nicely formatted string equivalent of the date and time.

To make directory information from the DTA more accessible to Turbo Pascal programs, I have defined a different record type providing better Pascal representation of DTA information, and pointers for use in linked lists:

```
DIRRec = RECORD
           FileName  : String15;
           Attrib    : Byte;
           FileSize  : LongInt;
           TimeStamp : TimeRec;
           DateStamp : DateRec;
           Prior     : DIRPtr;
           Next      : DIRPtr;
         END;
```

This record definition is included in the listings diskette for this book under the name **DIRREC.DEF.** The reserved DOS work area in the DTA has been omitted. The file name field is now a Pascal string with a length byte. The time and date stamps have been replaced with the time and date records described in Section 20.6. A separate time and date stamp are present in these records, but have been expanded to individual values for hours, minutes, and seconds; months, day, and year, and English-reading string representations of time and date.

The two pointer fields allow doubly linked lists of these records to be constructed on the heap. I'll be describing this process in considerable detail in Section 21.4.

Converting between the DTA format and the **DIRRec** format is accomplished by a single routine, **DTAToDIR**. The routine is not especially subtle.

```
{->>>>DTAtoDIR<<<<------------------------------------------}
{                                                          }
{ FILENAME DTATODIR.SRC -- Last modified 11/8/87           }
{                                                          }
{ This procedure converts data as returned by DOS FIND     }
{ calls $4E & $4F in the Disk Transfer Area (DTA) to a more  }
{ tractable form as defined by my own record type DIRRec.   }
{ This involves converting the time from a word timestamp to  }
{ a TimeRec, and the date from a word datestamp to a DateRec.  }
{                                                          }
{ DTAToDIR requires the prior definition of types DIRRec,    }
{ DIRPtr, and DTAPtr; and procedures CalcTime and CalcDate.   }
{                                                          }
{                                                          }
{                                                          }
{----------------------------------------------------------}

PROCEDURE DTAtoDIR(VAR OutRec : DIRRec);

VAR
  DTData      : DateTime;   { This type imported from DOS unit }
  I           : Integer;
  InRec       : SearchRec;  { Also imported from the DOS unit  }
  RegPack     : Registers;  { Also imported from the DOS unit  }
  CurrentDTA  : DTAPtr;

BEGIN
  RegPack.AX := $2F00; { Find current location of DTA }
  MSDOS(RegPack);
  WITH RegPack DO CurrentDTA := Ptr(ES,BX);
  InRec := CurrentDTA^;
  UnpackTime(InRec.Time,DTData);
  WITH OutRec DO                 { Now extract and reformat data }
    BEGIN
      FileName := InRec.Name; { Extract the file name }
      Attrib := InRec.Attr;   { Extract the attribute field }
      WITH TimeStamp DO       { Expand integer time stamp }
        BEGIN
          TimeComp := InRec.Time SHR 16;
          Hours := DTData.Hour;
          Minutes := DTData.Min;
          Seconds := DTData.Sec;
          Hundredths := 0;
        END;
      CalcTime(TimeStamp);    { Fill in the other time fields }
      WITH DateStamp DO       { Expand integer date stamp }
        BEGIN
          DateComp := InRec.Time AND $0000FFFF;
          Day := DTData.Day;
          Month := DTData.Month;
          Year  := DTData.Year;
        END;
      CalcDate(DateStamp);    { Fill in the other date fields }
      FileSize := InRec.Size;
      Next := NIL;             { Initialize the "next" pointer }
      Prior := NIL;            { Ditto the "prior" pointer }
    END
END;  { DTAtoDIR }
```

One point of interest is that you don't explicitly pass **DTAToDIR** a **SearchRec** to act as input. **DTAToDIR** gets its input in a somewhat unusual fashion, by querying DOS (by way of DOS function call $2F) to find out where the current DTA is, and assigns the address returned from DOS in ES : BX to a pointer. The DTA information is then accessed through that pointer.

It didn't have to be done that way, but I looked upon it as some guarantee that only the most current DTA would get converted (as encouragement to get information out of it and be done with the DTA format) and also as a demonstration of a DOS function call I might not otherwise be able to demonstrate. If you intend to keep **SearchRecs** around and need to convert them regardless of whether or not they are set as the current DOS DTA, it's trivial to pass a **SearchRec** as a value parameter.

DTAToDIR uses the **CalcDate** and **CalcTime** procedures from Section 20.6 to fill in the time and date stamp records in the **DIRRec** returned to the calling logic.

From Directory to String

Because directory information needs to be displayed to the user in a great many applications, it would be handy to have a further conversion between information as stored in a **DIRRec** record and a string easily displayable via **Write** or **Writeln**. Such a conversion function is provided by the string function **DIRToString**.

```
{->>>>DIRToString<<<<----------------------------------------}
{                                                            }
{ Filename : DIRSTRIN.SRC -- Last Modified 6/9/88            }
{                                                            }
{ This routine returns a String value containing all the     }
{ significant information from a directory record, formatted }
{ in a fashion similar to that used by the DOS DIR command   }
{ when it displays a file and its information.  A typical    }
{ string returned by DIRToString would look like this:       }
{                                                            }
{       DIRSTRIN.BAK 1697  01/07/87   3:04p                  }
{                                                            }
{ Type DIRRec must be predefined.                            }
{                                                            }
{                                                            }
{                                                            }
{------------------------------------------------------------}

FUNCTION DIRToString(InputDIR : DIRRec) : String;

CONST
  Blanker = '                                              ';

VAR
  Temp,WorkString : String80;
  DotPos : Integer;

BEGIN
  WITH InputDIR DO
```

```
    BEGIN
      Temp := '                                         ';
      {If the entry has the directory attribute, format differently: }
      IF (Attrib AND $10) <> 0 THEN    { Bit 4 is the directory attribute }
        BEGIN
          Insert(FileName,Temp,1);  { No extensions on subdirectory names }
          Insert('<DIR>',Temp,14)   { Tell the world it's a subdirectory  }
        END
      ELSE
        {This compound statement separates the file from its extension  }
        { and converts the file size to a string.  Note that we did not }
        { insert a file size figure into Temp for subdirectory entries. }
        BEGIN
          DotPos := Pos('.',FileName);
          IF DotPos > 0 THEN  { File name has an extension }
            WorkString := Copy(FileName,1,DotPos-1) +
              Copy(Blanker,1,9-DotPos) + '.' +
              Copy(FileName,DotPos+1,Length(FileName)-DotPos)
          ELSE
            WorkString := FileName + Copy(Blanker,1,8-Length(FileName)) + '.';
          Insert(WorkString,Temp,1);
          Str(FileSize:7,WorkString);
          Insert(WorkString,Temp,15)
        END;
      WITH DateStamp DO
        BEGIN
          { This is what it takes to assemble three separate integer  }
          { figures for month, day, and year into a string equivalent.}
          IF Month < 10 THEN Insert('0',DateString,1);
          IF Day < 10 THEN Insert('0',DateString,4);
          Insert(DateString,Temp,24);
        END;
      Insert(TimeStamp.TimeString,Temp,34); { Finally, insert the time }
    END;
  Delete(Temp,42,Length(Temp)-42);
  DIRToString := Temp
END;
```

There's nothing especially subtle about **DIRToString**; it fills a string with information plucked from the **DIRRec** passed in parameter **InputDIR**. For a typical directory entry of a DOS file, the string returned by **DIRToString** looks like this:

```
DIRSTRIN.BAK      2623  01/13/87    7:53p
```

This string value returned by **DIRToString** is exactly 39 characters long, meaning that two such strings can be displayed side-by-side on an 80-column screen, with two spaces between them in the middle.

DIRToString will play a key role in two utility programs described later in this book: **Locate** and **Spacer**.

Search and Search Again

On the surface of it, you'd think that the simplest method possible for searching for a given file would simply be to attempt to open it. However, that ignores the possibility of using "ambiguous" file names, that is, file names containing "wild card" characters that can match one file or many files. These file names are valid at the keyboard, and they are valid from within a program as well. For example, making use of ambiguous file names would allow your program to identify every file on a disk with an extension of .PAS, by using the ambiguous file name "*.PAS", and work only with those files.

There are two kinds of wild card characters: ? and *. The question mark character replaces one character in a file name, and no more; in other words, MODE?ERR.MSG would apply equally to MODE1ERR.MSG, MODE2ERR.MSG, MODE3ERR.MSG, and so on. The asterisk means "I match anything from here to the end." The "end" here is either the dot character, for the main portion of the file name, or the end of the file name proper, for the file name extension. In other words, MODE*.MSG would match all of the three file names mentioned earlier in this paragraph, and the familiar *.* matches anything at all.

DOS's mechanism for dealing with one file specification matching many files involves two separate DOS function calls: FIND FIRST ($4E) and FIND NEXT ($4F). It works like this: You assemble a *file spec* for the file or files you wish to locate. A file spec is the string containing a full file specifier including the drive unit and path name. For example, while "LOCATE.PAS" is a file name, its full file spec might be something like "D:\TURBO\HACKS\LOCATE.PAS." You pass this file spec to FIND FIRST, and it will locate the first file matching that file spec if one exists, or return an error message if there is no match at all. If at least one is found, you then can call FIND NEXT repeatedly, and FIND NEXT will keep returning matching directory entries in the DTA until no more files match the file spec. Then an appropriate error code is returned in AX.

At first glance, it seems odd that two DOS function calls would be required to do this. A single FIND call could both begin the search and keep searching until it found all directory entries matching the file spec passed to it. DOS's two-call system is actually very efficient, since FIND FIRST does all the setting up of the file spec, and arranges some special information in the DTA to make the search possible. Once that setup has been done once, it need not be done again, and if the DTA is not disturbed, FIND NEXT need only continue the search, and not bother setting up all the search machinery for each additional search on the same file spec.

The FindFirst and FindNext Procedures

Turbo Pascal hides much of the messiness of using the FIND FIRST and FIND NEXT DOS function calls by embedding them within two procedures in the **DOS** unit: **FindFirst** and **FindNext**.

Setting up the DTA (in the form of a **SearchRec**) and performing the first search for a matching file is done by **FindFirst**:

```
PROCEDURE FindFirst(Spec : String; Attr : Word; VAR F : SearchRec);
```

The full file spec including the path is passed in **Spec**. **Spec** may be ambiguous; in other words, it may contain wild card characters. If no directory information is included, the current directory is searched. **Attr** contains the attribute byte that is to apply to the search. There is some trickiness involved here, as I'll explain in connection with searching for volume labels and subdirectories a little later. Finally, **F** is the DTA to be used for the search.

You can tell if a **FindFirst** call has been successful in locating a matching file by checking the **DOSError** variable declared in the **DOS** unit. A 0 returned in **DOSError** indicates that the search was successful, and that the DTA contains the first matching file located. Remember that that may not be the *only* file to match the file spec, since the file spec may contain wild card characters. Any other value indicates some sort of error. Errors to watch for are:

```
2  ($02)  File not found
3  ($03)  Path not found
18 ($12)  No more files to be found
```

Error 3 occurs when you pass a subdirectory name as part of **Spec** and DOS can't find that subdirectory. Other errors may occur under special circumstances, but these three are the most common. A successful search will leave **DOSError** set to 0 and present you with a **SearchRec** filled with information about the found file.

Now, as we said, finding a file through **FindFirst** is the last word *only* if the file spec passed in **Spec** was unambiguous, in other words, if it contains no wild card characters. If you're searching for a group of files (perhaps all of those ending in .PAS) you need to process the information returned by **FindFirst** and continue the search. Continuing is done with **FindNext**:

```
PROCEDURE FindNext(VAR F : SearchRec);
```

Here, **SearchRec** is the *same* **SearchRec** you passed to **FindFirst** during the first part of the search. Keep in mind that you cannot call **FindNext** without first having called **FindFirst**. Furthermore, the call to **FindFirst** must have been successful; you must have received a 0 in **DOSError**.

I must emphasize that you cannot change search specs in the middle of the search by tinkering with **SearchRec** parameter **F**. For example, you can't decide to change the file attribute during repeated calls to **FindNext** by changing the **Attr** field in **F**. Once you begin a search with **FindFirst**, the only way to change the search conditions is to execute **FindFirst** again with the new spec and attribute byte. At that point, of course, it becomes a whole new search.

FindNext can be called repeatedly after the initial call to **FindFirst**, and the process can continue until error 2 or error 18 comes back in **DOSError**. At this point it's fruitless to go on, since DOS has failed to find any further matching files.

20.8: CREATING AND READING VOLUME LABELS

Reading volume labels provides an interesting first exercise in the use of **FindFirst**. Before we get into that, some discussion of volume labels and their role in PC DOS must come first.

Volume labels are a tip of the DOS hat to mainframe operating systems, which have long provided for the application of a unique machine-readable label to a storage volume, be it removable disk pack, fixed disk pack, or magnetic tape. Anyone can stick an adhesive label on a diskette, of course, but volume labels allow both the human user and the application program to "read" the label on the diskette. That way, if the program directs the user to "Insert the ACCOUNTS diskette in drive A:" the program can immediately determine if the right diskette made it into the drive without having to poll the names of the files on the disk to make sure.

PC DOS first supported volume labels with V2.0, but the support in DOS 2.X is limited and crude. My recommendation is that you do *not* try to build volume label support into your applications unless they can be guaranteed to run only under DOS 3.X or higher.

For that reason, this particular section makes the same assumption, that the **CreateLabel** and **GetLabel** routines will only be run under DOS 3.X. I have not tested them under DOS 2.X and cannot guarantee that they will work under all circumstances for DOS 2.X.

A volume label is physically nothing more than an empty file (that is, a file with 0 bytes in it) with the volume label attribute bit set to 1 (see Figure 20.6). All directory entry fields aside from creation time and date, file name, and attribute byte are zeroed out. DOS will only set attribute bit 3 (the Volume Label bit) on a file created in the root directory. By going beneath the level of DOS it is possible to set the Volume Label bit on a file in a subdirectory, of course, but how DOS would deal with such a file, I couldn't say.

It's also possible to set the Volume Label bit on multiple files in the root directory, but only the one having the physically first directory entry will be treated as the real DOS volume label.

Reading and creating volume labels is easy. Changing or deleting them is *not*. Once a directory entry has had its Volume Label bit raised to 1, none of the familiar DOS function calls will touch it, and only FIND FIRST will be able to detect its presence, and then only if it's the *first* directory entry with the bit set. Whether this is a bug due to Microsoft's forgetfulness or a feature supporting media security is unclear; that depends on how much you love Microsoft or how paranoid you are. I lean toward the former view; it seems silly to restrict the modification of volume labels so severely when *creating* them is so simple.

So once again, care must be counseled. Before you create a volume label for a disk volume, make triple sure the label you're creating is the label you want. The only safe recourse to an undesired label is to reformat the disk and recreate the label in the process.

There have been, of course, published methods of deleting and changing volume labels by locating the directory sectors on the disk and physically altering the sectors to change the directory entry tagged as the volume label. This operates beneath the level at which DOS wants you to work, and there's more than just "sticking to the rules" at stake here. The number of disk storage devices has grown explosively in the last few years, and some of them (the Bernoulli Box comes to mind) are significantly different from either floppy disks or traditional hard disks. I have not yet seen any truly reliable way of determining where the directory physically begins on the disk for any arbitrary disk type, and when you're talking about writing to sectors involving the disk directory or FAT, *you do not want to get it wrong.* If your application tries to alter the volume label on a customer's 200MB odd-format Winchester disk drive and scrambles it in the process, you will at very least lose the customer and possibly hear from his lawyers as well, because the likelihood of a disk device being fully backed up at all times varies inversely with its capacity.

For this reason, the two volume label routines presented here either read or create a volume label, but do not alter or delete volume labels that already exist. My hunch is that future releases of DOS will correct this oversight, since history has shown that volume label support has grown better as DOS has evolved. Furthermore, when you move to operating systems like OS/2 that support multitasking and multiple users, the OS must by need get very very hardnosed about keeping applications from snooping critical system resources (like the FAT and disk directories) from under the table. The ability is not there now, but I don't think you'll have to wait very long for it.

Reading a Volume Label

Given that a DOS volume label is the first file in the root directory of a volume with bit 3 of the attribute byte set, the FIND FIRST DOS function call (as embodied in procedure **FindFirst**) is an intuitive means of reading a volume label. It is, in fact, the only means short of directly reading the directory sectors on the disk. The function **GetLabel** uses **FindFirst** to return the DOS volume label in the **SearchRec** record **F**.

```
{->>>>GetLabel<<<<--------------------------------------------}
{                                                             }
{ Filename : GETLABEL.SRC -- Last Modified 7/12/88            }
{                                                             }
{ This function returns a string value that is the volume     }
{ label of the drive passed in DriveSpec.  No check is made   }
{ as to the validity of the character in DriveSpec; if there  }
{ is no corresponding drive the system may hang or return an  }
{ error depending on the specifics.  If no volume label exists }
{ for the specified volume, a null string (zero length) will  }
{ be returned and parameter LabelFound will be set to FALSE.  }
{                                                             }
{                                                             }
{                                                             }
{-------------------------------------------------------------}


```

```
FUNCTION GetLabel(DriveSpec : String; VAR LabelFound : Boolean) : String;

TYPE
  String80 = String[80];

VAR
  I          : Integer;
  SearchSpec : String80;
  Temp       : String80;
  Regs       : Registers;
  ASCIIZ     : ARRAY[1..81] OF Char;
  DTA        : SearchRec;

BEGIN
  SearchSpec :=  DriveSpec + ':\*.*' + Chr(0);
  FindFirst(SearchSpec,$08,DTA);

  Temp := '';  { So we can return a null string if no label found }
  IF (DOSError = 2) OR (DOSError = 18) THEN     { Label not found }
    LabelFound := False
  ELSE
    BEGIN
      LabelFound := True;
      Temp := DTA.Name;
      { If a dot exists in the DTA file name, get rid of it: }
      IF Pos('.',Temp) > 0 THEN Delete(Temp,Pos('.',Temp),1);
    END;
  GetLabel := Temp;               { Assign function return value }
END;
```

FindFirst requires a *search spec*, which is the full pathname of the file to be searched for, including the drive specifier. In this case, the file name we want to locate is the familiar *.* which means any file name at all. If we don't know what the volume label is, we need to use a completely ambiguous file name. In other applications, we might want to find a file with a particular name, perhaps to see if it exists on the specified volume. In that case, the filename would be part of the search spec, rather than *.*.

The search spec is constructed in a string variable by concatenating the drive specifier and the pathname *.*, which means any filename in the root directory. In other applications, the search spec could include a path down through several levels of subdirectories to a specific file or *.* within a specific subdirectory.

The $08 value passed to **FindFirst** is a byte with only the volume label attribute bit (bit 3) set. This is how we direct **FindFirst** to locate a volume label directory entry and nothing else.

This exclusive treatment of the volume label attribute bit is a special case. The other attribute bits are treated very differently by FIND FIRST, as I will explain a little later when we start using **FindFirst** to look for ordinary files.

When DOS returns after **FindFirst** is invoked, two situations are possible: Either an error message comes back in **DosError**, or the DTA has been filled with a directory entry containing the DOS volume label on the specified volume.

DOS documentation states that the error message will be either 2 (File not found) or 18 (No more files to be found) but in practice I have never seen anything but error 18 returned under DOS 3.1. **GetLabel** itself does not return an error code to the calling logic; if the volume label was not found, **VAR** parameter **LabelFound** will return a value of **FALSE**.

One peculiarity about volume labels is that, although they are actually filenames, DOS displays them without splitting them into the traditional filename/file extension duo, divided by a period character. So when you retrieve a volume label from a DTA, you need to remove the period character *if* the volume label is longer than eight characters. In other words, the volume label CALIBAN will be found in the DTA as CALIBAN but the longer volume label WIDDERSHINS will be found as WIDDERSH.INS. One **Delete** statement does the job.

Creating a Volume Label

If a disk volume does not already have a label, you can create one with a single DOS function call. DOS function call $3C (CREATE) is used to create a file anywhere on a disk, given its full file spec including path. Function $3C does not actually write any information into the file. It simply creates a directory entry and fills out the time and date stamps, name, and attribute byte. The resulting file has a length of 0 and occupies no space on the disk other than the space taken by the directory entry. Furthermore, if the attribute byte passed to function $3C contains bit 3 set to 1, the created file will become the volume's volume label, if the volume in question doesn't already have a label.

```
{->>>>CreateLabel<<<<---------------------------------------}
{                                                           }
{ Filename : CREATLBL.SRC -- Last Modified 7/11/88          }
{                                                           }
{ This procedure creates a new volume label on the unlabeled }
{ DOS volume passed in DriveSpec.  No check is made         }
{ as to the validity of the character in DriveSpec; if there }
{ is no corresponding drive the system may hang or return an }
{ error depending on the specifics.  If a volume label already }
{ exists on the specified volume, CreatedLabel will return   }
{ FALSE with an ErrorReturn value of 0.  This is not really an }
{ error condition, but DOS makes no provision for altering a  }
{ volume label that already exists, so at best we go home with }
{ our tail between our legs.  If some sort of true error      }
{ occurs, the DOS error code will be returned in ErrorReturn,  }
{ and CreatedLabel will be set to FALSE.  If CreatedLabel     }
{ comes back TRUE, the label was in fact created.            }
{                                                           }
{ Function GetLabel must be predefined.                     }
{                                                           }
{                                                           }
{                                                           }
{-----------------------------------------------------------}

```

```
PROCEDURE CreateLabel(DriveSpec        : String;
                      NewLabel         : String;
                      VAR CreatedLabel : Boolean;
                      VAR ErrorReturn  : Word;
                      ShowError        : Boolean);

TYPE
  ErrorCode = 0..18;    { DOS function call error codes }
  String80  = String[80];

VAR
  I            : Integer;
  SearchSpec   : String80;
  FileSpec     : String80;
  CurrentLabel : String80;
  Regs         : Registers;
  ASCIIZ       : ARRAY[1..81] OF Char;
  DTA          : SearchRec;
  Error        : ErrorCode;
  FoundLabel   : Boolean;

BEGIN
  CurrentLabel := GetLabel(DriveSpec,FoundLabel);
  IF NOT FoundLabel THEN { No label exists yet }
    BEGIN
      FileSpec := DriveSpec + '\' + NewLabel + Chr(0);
      Move(FileSpec[1],ASCIIZ,Sizeof(FileSpec));

      WITH Regs DO
        BEGIN
          AH := $3C;          { $3C = Create File }
          DS := Seg(ASCIIZ);  { Put address of ASCIIZ }
          DX := Ofs(ASCIIZ);  { in DS : DX }
          CL := $08;          { Set Volume Label attribute }
        END;
      MSDOS(Regs);                    { Make CHMOD DOS call }

      { If the Carry Flag is found to be set, it's an error: }
      IF (Regs.Flags AND $0001) = 1 THEN
        BEGIN
          CreatedLabel := False;   { No luck }
          ErrorReturn := Regs.AX;  { Return error code as parameter }
          Error := ErrorReturn;    { Make an ordinal of the error code }
          IF ShowError THEN
            CASE Error OF
              2 : Writeln('Label file not found.');
              3 : Writeln('Bad path error -- possible disk failure.');
              5 : Writeln('Access to label denied -- Disk write protected?');
              ELSE Writeln('Unexpected DOS error ',Error,' on label write.')
            END; { CASE }
        END
      ELSE CreatedLabel := True   { No error - created the label }
    END
  ELSE     { Label already exists; can't re-create it... }
    BEGIN
      CreatedLabel := False;
      ErrorReturn := 0;
    END
END;
```

DOS function call $3C is used to create a volume label in the procedure **CreateLabel**. To determine if a label already exists on the requested volume, **CreateLabel** calls **GetLabel**. If a label already exists, the Boolean variable **CreateLabel** is returned to the calling logic set to **FALSE**. If no label is found, **CreateLabel** sets up function call $3C with $08 in CL. Again, $08 is the numeric value of the attribute byte with bit 3, the volume label bit, set to 1.

CreateLabel gives you the option of returning an error code without any visible display of an error message, or actually displaying an error message on the console for the user.

Ordinarily, you wouldn't need to use DOS function $3C to create a file, since you can create a 0-length file simply by using **Assign** to assign a file variable to a pathname and **Rewrite** to write the directory entry to the disk. **Assign** and **Rewrite** do not, however, give you access to the attribute byte, which is why function $3C must be used to create a volume label.

20.9: SEARCHING FOR GROUPS OF FILES

FindFirst will locate the first file matching a given file spec. DOS has a streamlined method to take it from there: Once **FindFirst** has set up the necessary criteria, the FIND NEXT function call (as embodied in the **FindNext** procedure in the **DOS** unit) will repeatedly search the specified path until no more files are found that match the file spec specified to **FindFirst**.

After calling the **FindNext** procedure, when control returns to your Turbo Pascal program, either a found directory entry will be in **FindNext**'s **SearchRec** parameter (which is simply a DTA mapped upon a Pascal record) or an error code will be in **DOSError** indicating that no more files are to be found on that path.

The example program I've written to illustrate the use of **FindFirst** and **FindNext** is a truly useful one, now that 20- and 30-megabyte hard disks are common and 80-megabyte hard disks are not out of reach for many of us. If it hasn't already, a situation like this will soon arise: Somewhere on your hard disk you suspect there is a public doman utility program called FASTVID.COM—or was it FASTV.COM? Or FASTVID.EXE? Or were you imagining it all along? With 500 files scattered across 40 nested subdirectories, it could take quite a bit of searching to locate the mystery utility.

Or you could let the computer do what it does best and find it for you. This is the purpose of the program **Locate**, which I will be describing shortly. **Locate** searches any directory and all its child subdirectories for files that match a given file specification, including wildcards. If no directory is given, **Locate** will search the entire disk.

Locate is a more difficult program than most to understand because it operates recursively. Some people just have a hard time dealing with recursion, and if you're one of the people who do, **Locate** will have that uncomfortable feeling of black magic about it.

Recursive Tree Search

You may have heard the term *tree-structured directories* in reference to DOS V2.X subdirectory structures, particularly when there are several layers of nested subdirectories involved. The "tree" metaphor stems from the fact that one "root" directory can have any number of "branch" directories, each of which can itself have more "branch" directories, and so on. The directory structure "spreads out" from one single root directory into a structure reminiscent of a tree.

Figure 20.7 is a slightly simplified diagram of a system of nested subdirectories. The top directory is the root, with each vertical partition representing one directory entry. Directory entries are either files or subdirectories. Each subdirectory entry in a directory points to its own subdirectory proper, and subdirectories may be nested in this fashion as deeply as desired.

The single-dot and double-dot symbols should be familiar from DIR listings of subdirectories. The single dot represents the current directory, and the double dot the parent directory. Why directory entries are required for the current and parent directories is complex and has to do with the way DOS keeps track of disk storage. In a sense, a tree-structured directory is a doubly linked list of directories, but if that makes no sense to you, forget it; it is not germane to the current discussion. Note that the root directory has entries neither for parent directory, obviously, nor for itself; this is one way in which the root directory is special and distinct from subdirectories.

Searching a tree structure like this recursively can best be explained by following the search path on a diagram (Figure 20.8). For the sake of discussion, imagine a Pascal procedure named **SearchDirectory**. You pass it a directory specifier and a file name, and it searches that directory and any of its child directories for anything matching that file name, printing out the name and full path of any file it finds.

The search begins at the root directory. **SearchDirectory** is called and given the root directory specifier (the solo backslash \) as the directory to search. Directory entries are examined one by one to see if the file names match the search spec. This prosaic process continues until **SearchDirectory** encounters a subdirectory entry; let's call it subdirectory **TURBO**. **TURBO** needs to be searched too. This is where recursion happens.

SearchDirectory calls itself. What happens here is that the current "state" of **SearchDirectory** is pushed on the stack, and an entirely new copy of **SearchDirectory**'s local variables and parameters is created. This is called an *instantiation* of **SearchDirectory**. The new "copy" of **SearchDirectory** is passed the root directory specifier \ *plus* the name of the new subdirectory, **TURBO**. It then begins searching directory \TURBO for file matches.

As it did with the root directory, **SearchDirectory** examines the directory entries in **\TURBO** one by one for matches. It displays any it finds. Along the way it discovers another subdirectory, a child directory to **\TURBO** called **HACKS**. **HACKS** needs to be searched, just as any other directory does. **SearchDirectory** calls itself once again.

This time, the parent directory specifier **\TURBO** is prefixed to **HACKS** (with a backslash separator) making the new directory specifier **\TURBO\HACKS**. A third

Figure 20.7

Tree-Structured Directories

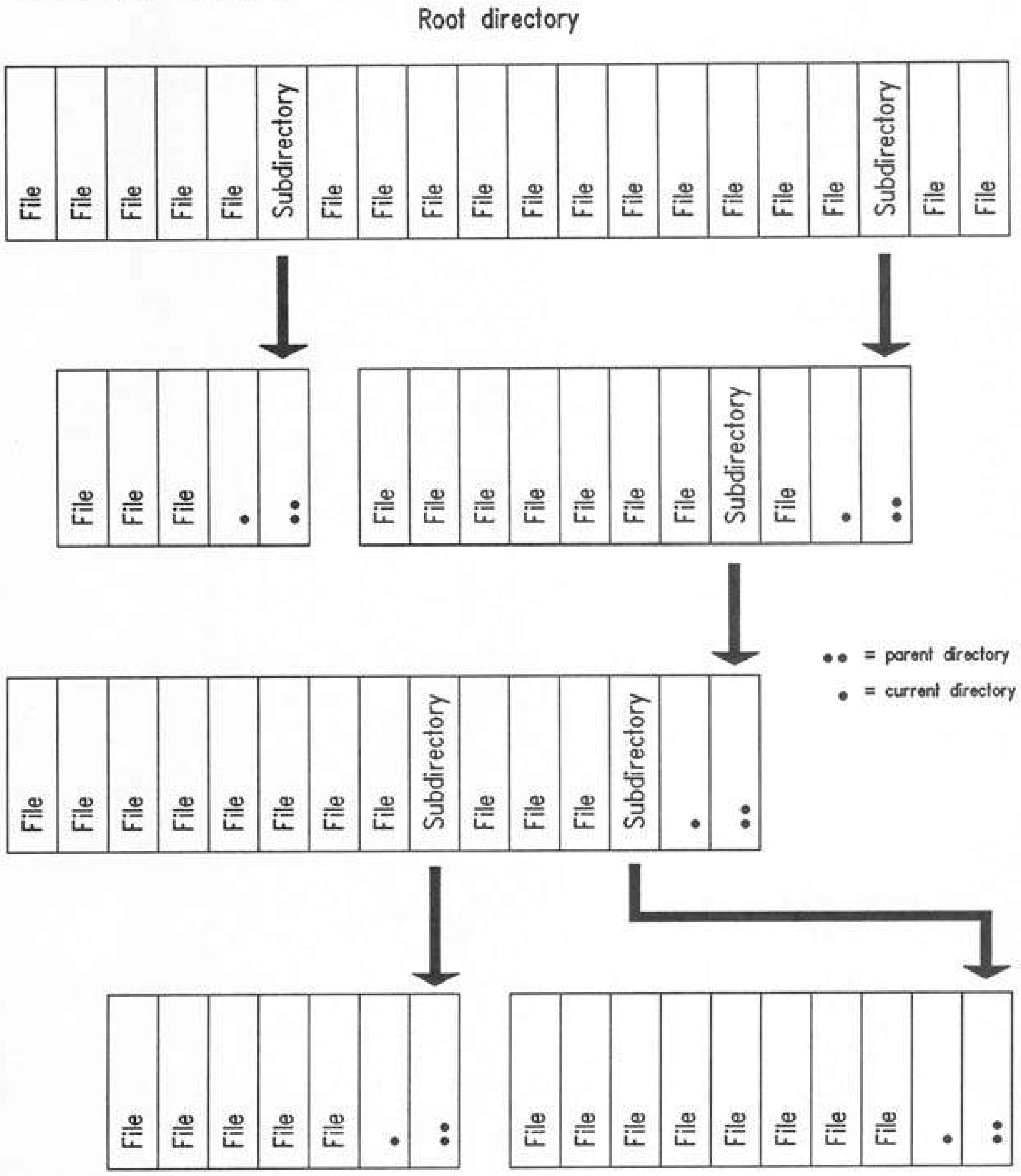
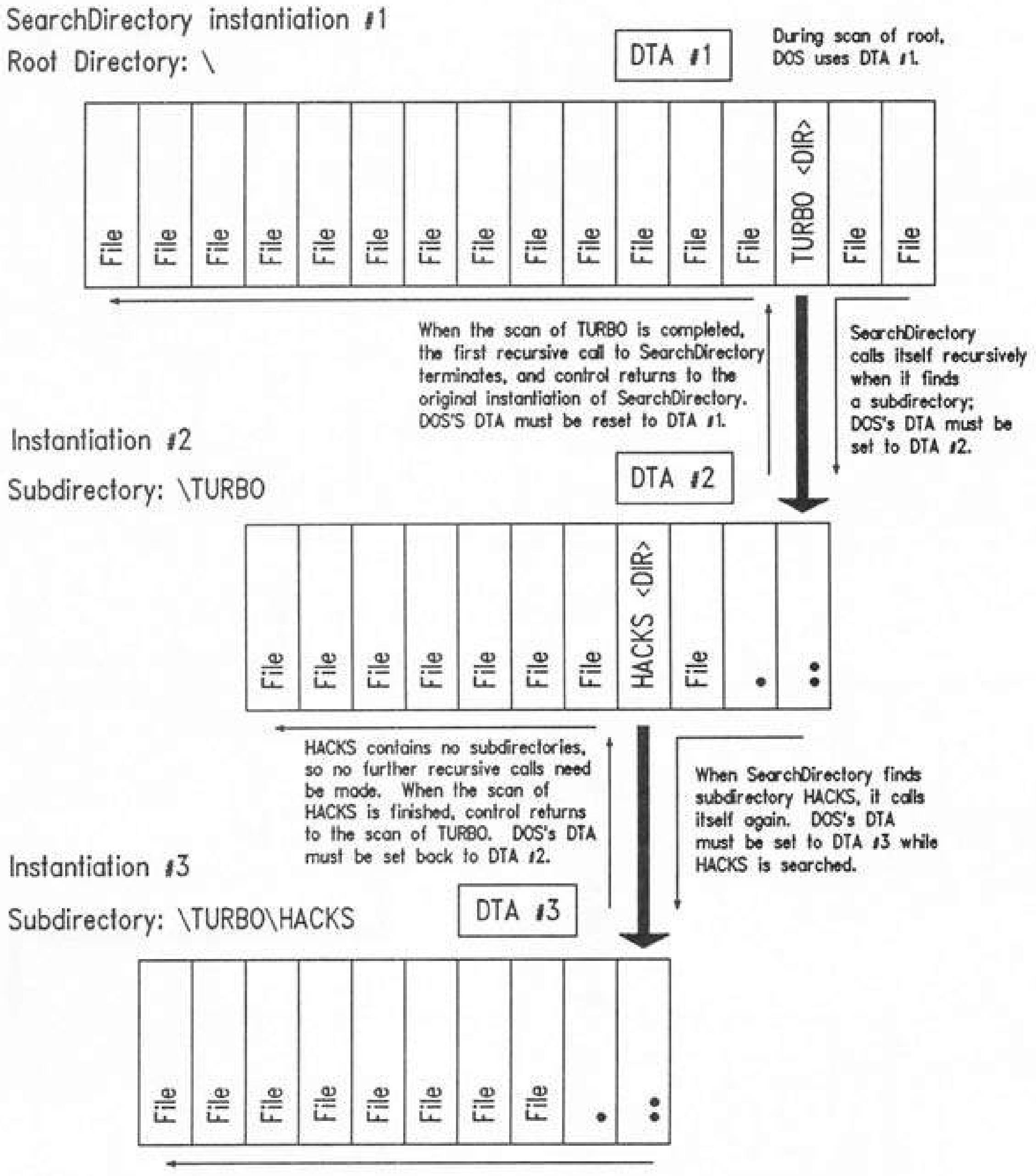


Figure 20.8

Recursive Directory Search


SearchDirectory instantiation #1
Root Directory: \
DTA #1
During scan of root, DOS uses DTA #1.
File
TURBO <DIR>
When the scan of TURBO is completed, the first recursive call to SearchDirectory terminates, and control returns to the original instantiation of SearchDirectory. DOS'S DTA must be reset to DTA #1.
SearchDirectory calls itself recursively when it finds a subdirectory; DOS's DTA must be set to DTA #2.
Instantiation #2
Subdirectory: \TURBO
DTA #2
HACKS <DIR>
.
..
HACKS contains no subdirectories, so no further recursive calls need be made. When the scan of HACKS is finished, control returns to the scan of TURBO. DOS's DTA must be set back to DTA #2.
When SearchDirectory finds subdirectory HACKS, it calls itself again. DOS's DTA must be set to DTA #3 while HACKS is searched.
Instantiation #3
Subdirectory: \TURBO\HACKS
DTA #3

instantiation of **SearchDirectory** is created after the second has been safely pushed onto the stack. Again, directory entries are examined one by one for file matches, with an eye out for more subdirectories.

HACKS, however, has no subdirectories. When the scan of **HACKS** is completed, the third instantiation of **SearchDirectory** terminates. As with any Pascal subprogram that finishes execution, it returns control to the calling logic. In this case, the calling logic was the second instantiation of **SearchDirectory**. The second instantiation is popped from the stack and begins running again, checking directory entries in **\TURBO** for file matches as though nothing had interrupted it.

Subdirectory **\TURBO**, as it happens, has no more subdirectories. If it had, they would have been searched in exactly the same way **HACKS** was searched, via another recursive call to **SearchDirectory**. So the rest of **\TURBO** is searched for matching files. Eventually the search is completed. The second instantiation of **SearchDirectory** terminates, and returns control to the first instantiation, the one searching the root directory. The first instantiation of **SearchDirectory** is popped off the stack, and the search of the root directory continues. Once the scan of the root is completed, the original instantiation of **SearchDirectory** terminates, having searched the entire disk for files matching the given file spec.

From 10 steps back, this is all that happens. Beneath the surface, however, is an additional detail that has to be tended to: the current DTA.

The actual search mechanism used by **SearchDirectory** is DOS's FIND FIRST and FIND NEXT function calls (through the **FindFirst** and **FindNext** procedures), which require the use of the DOS DTA through a Pascal record called a **SearchRec**. FIND NEXT works only over a single directory, and each directory being searched needs its own exclusive use of a DTA. From a Pascal perspective this is no problem. Pascal creates a new **SearchRec** called **CurrentDTA** on the stack for each new instantiation of **SearchDirectory**. *However, DOS doesn't "know" that it must switch to the new instantiation's* ***SearchRec****.* DOS, in a sense, isn't in on the recursion process, which is handled entirely by the Turbo Pascal runtime code.

Therefore, each time a call to **FindFirst** or **FindNext** is made, DOS must be "told" which DTA to use. You don't have to worry about that part of the problem, since **FindFirst** and **FindNext** each contain a call to DOS function $1A, which sets the address of the current DTA.

It's enough to make your head spin. The process is best followed along on a diagram; again, trace through the three instantiations of **SearchDirectory** in Figure 20.8. Each directory needs its own DTA for the search process, and the three DTA's are numbered from one to three. DTA 1 is used during the search of the root directory. When the first recursive call to **SearchDirectory** is made, an explicit call to DOS function $1A is made within **FindFirst**, changing the current DTA (that is, the one DOS will use) to DTA 2. DTA 2 remains the current DTA as long as the second instantiation of **SearchDirectory** has control. But when the third instantiation is created to search the third directory **\TURBO\HACKS**, the current DTA must be changed yet again, to DTA 3. DTA 3 remains the current DTA as long as subdirectory **\TURBO\HACKS** is being searched.

\TURBO\HACKS is the bottom of the tree. From that point we start going home, and the whole process of changing DTA's as we wove our way downward through the directory tree must be undone as we move back upward again. The first change happens as we leave the third instantiation behind. DTA 3 was in force during the search of **\TURBO\HACKS**; now we are returning to the search of **\TURBO**, and we need to go back to DTA 2. (DTA 2 was waiting on the stack, safe and sound, and still contains setup information from the **FindFirst** call that began the search of **\TURBO**.) When the search of **\TURBO** is finished, DTA 2's job is done. Control returns to the first instantiation of **SearchDirectory** to finish out the search of the root directory, and the current DTA must be changed again from DTA 2 to DTA 1, again on the stack, waiting to go back to work and finish the **FindNext** sequence that inspects directory entries one by one.

Each time we need to change *back* from the DTA used during a recursive call to **SearchDirectory**, a call must be made to DOS function $1A. Again, this is done safely within **FindNext**, but if you were working with raw DOS calls through Turbo Pascal's **MSDOS** procedure, you would have to call function **$1A** yourself, as did the earlier version of **Locate** in my book *Turbo Pascal Solutions* (Scott, Foresman, 1987).

Program Locate

With all of that convolution safely beneath your belt, take a long, close look at the actual code for program **Locate**. The entirety of the **Locate** program is only a frame for procedure **SearchDirectory**, which does the only real work in **Locate**. The search process was built into a procedure so that it could call itself, because programs in Pascal cannot call themselves recursively.

```
{--------------------------------------------------------------}
{                             LOCATE                           }
{                                                              }
{             Disk file tree-search search utility             }
{                                                              }
{                               by Jeff Duntemann              }
{                               Turbo Pascal V5.0              }
{                               Last update 5/22/88            }
{                                                              }
{ This utility searches a tree of directories (from the root   }
{ or from any child directory of the root) for a given file    }
{ spec, either unique or ambiguous.  It provides a good        }
{ example of the use of the DOS 2.X/3.X FIND FIRST/NEXT        }
{ function calls.  See the main program block for instructions }
{ on its use.                                                  }
{                                                              }
{                                                              }
{                                                              }
{--------------------------------------------------------------}

PROGRAM Locate;

```

```
USES DOS;

TYPE
  String80 = String[80];
  String15 = String[15];

{$I TIMEREC.DEF}   { Described in Section 20.6 }
{$I DATEREC.DEF}   { Described in Section 20.6 }
{$I DIRREC.DEF}    { Described in Section 20.7 }

  DTAPtr   = ^SearchRec;


VAR
  I,J               : Integer;
  SearchSpec        : String80;
  InitialDirectory  : String80;
  Searchbuffer      : SearchRec;


{$I DAYOWEEK.SRC}  { Described in Section 20.6 }
{$I CALCDATE.SRC}  { Described in Section 20.6 }
{$I CALCTIME.SRC}  { Described in Section 20.6 }
{$I DIRSTRIN.SRC}  { Described in Section 20.7 }
{$I DTATODIR.SRC}  { Described in Section 20.7 }


{->>>>SearchDirectory<<<<------------------------------------}
{                                                            }
{ This is the real meat of program LOCATE.  The machinery    }
{ for using FIND FIRST and FIND NEXT are placed in a procedure }
{ so that it may be recursively called.  Recursion is used   }
{ because it is the most elegant way to search a tree, which }
{ is really all we're doing here.  All the messiness (and it }
{ IS messy!) exists to cater to DOS's peculiarities.         }
{                                                            }
{ For example, note that each recursive instantiation of     }
{ SearchDirectory needs its own DTA.  No problem--one is     }
{ created on the stack each time SearchDirectory is called.  }
{ BUT--DOS is not a party to the recursion, so the DTA address }
{ must be set both before AND after the recursive call, so   }
{ that once control comes BACK to an instance of             }
{ SearchDirectory that has been left via recursion, DOS can  }
{ "come back" to the temporarily dormant DTA, which may still }
{ contain information necessary to execute a FIND NEXT call. }
{                                                            }
{ Much of the rest of the fooling around involves formatting }
{ the search strings correctly for passing to the next       }
{ instantiation of SearchDirectory.                          }
{                                                            }
{ It's not documented, but I have found that DOS returns error }
{ code 3 (Bad Path) on a file FIND when the path includes a  }
{ nonexistant directory name.  Error code 2, on the other    }
{ hand, while documented, never seems to come up at all.     }
{------------------------------------------------------------}


PROCEDURE SearchDirectory(Directory,SearchSpec : String);
```

```

VAR
  NextDirectory : String;
  TempDirectory : String;
  CurrentDTA    : SearchRec;
  CurrentDIR    : DIRRec;
  Regs          : Registers;


{>>>>DisplayData<<<<}
{ Displays file data and full path for the passed file }

PROCEDURE DisplayData(Directory : String; CurrentDIR : DIRRec);

VAR
  Temp : String;

BEGIN
  Temp := DIRToString(CurrentDIR);
  Delete(Temp,1,13);
  Write(Temp,Directory);
  IF Directory <> '\' THEN Write('\');
  Writeln(CurrentDIR.FileName);
END;



BEGIN
  { First we look for any subdirectories.  If any are found, }
  { we make a recursive call and search 'em too: }

  { Suppress unnecessary backslashes if we're searching the root: }
  IF Directory = '\' THEN
    TempDirectory := Directory + '*.*'
  ELSE
    TempDirectory := Directory + '\*.*';

  { Now make the FIND FIRST call for directories: }

  FindFirst(TempDirectory,$10,CurrentDTA);


  { Here's the tricky stuff.  If we get an indication that there is }
  { at least one more subdirectory within the current directory,    }
  { (indicated by lack of error codes 2 or 18) we must search it    }
  { by making a recursive call to SearchDirectory.  We continue     }
  { recursing and returning from the searched subdirectories until  }
  { we get a code indicating none are left. }
  WHILE (DOSError <> 2) AND (DOSError <> 18) DO
    BEGIN
      IF  ((CurrentDTA.Attr AND $10) = $10)   { If it's a directory }
      AND (CurrentDTA.Name[1] <> '.') THEN  { and not '.' or '..' }
        BEGIN
          { Add a slash separating sections of the path if we're not }
          { currently searching the root: }
          IF Directory <> '\' THEN NextDirectory := Directory + '\'
            ELSE NextDirectory := Directory;

          { This begins with the current directory name, and copies }
          { the name of the found directory from the current DTA to }
          { the end of the current directory string.  Then the new  }
          { path is passed to the next recursive instantiation of   }
          { SearchDirectory. }
          NextDirectory := NextDirectory + CurrentDTA.Name;

          { Here's where we call "ourselves." }
          SearchDirectory(NextDirectory,SearchSpec);

        END;
       FindNext(CurrentDTA);  { Now we look for more... }
    END;

  { Now we can search for files, once we've run out of directories.  }
  { This is conceptually simpler, as recursion is not involved.      }
  { We combine the path and the file spec into one string, and make  }
  { the FIND FIRST call: }

  { Suppress unnecessary slashes for root search: }
  IF Directory <> '\' THEN
    TempDirectory := Directory + '\' + SearchSpec
  ELSE TempDirectory := Directory + SearchSpec;

  { Now, make the FIND FIRST call: }
  FindFirst(TempDirectory,$07,CurrentDTA);

  IF DOSError = 3 THEN       { Bad path error }
    Writeln('Path not found; check spelling.')

  { If we found something in the current directory matching the filespec, }
  { format it nicely into a single string and display it: }
  ELSE IF (DOSError = 2) OR (DOSError = 18) THEN
    { Null; Directory is empty }
  ELSE
    BEGIN
      DTAtoDIR(CurrentDIR);      { Convert first find to DIR format.. }
      DisplayData(Directory,CurrentDIR);        { Show it pretty-like }

      IF DOSError <> 18 THEN { More files are out there... }
        REPEAT
          FindNext(CurrentDTA);
          IF DOSError <> 18 THEN  { More entries exist }
            BEGIN
              DTAtoDIR(CurrentDIR); { Convert further finds to DIR format }
              DisplayData(Directory,CurrentDIR)         { and display 'em }
            END
        UNTIL (DOSError = 18) OR (DOSError = 2)  { Ain't no more! }
    END
END;


BEGIN
  IF ParamCount = 0 THEN
    BEGIN
      Writeln('>>LOCATE<<  V2.00  By Jeff Duntemann');
      Writeln('              From the book, COMPLETE TURBO PASCAL 5.0');
      Writeln('              Scott, Foresman & Co. 1988');
```

```
        Writeln('            ISBN 0-673-38355-5');
        Writeln;
        Writeln('This program searches for all files matching a given ');
        Writeln('filespec on the current disk device, in any subdirectory.');
        Writeln('Now that 32MB disks are getting cheap, we can pile up');
        Writeln('great heaps of files and easily forget where we put things.');
        Writeln('Given only the filespec, LOCATE prints out the FULL PATH');
        Writeln('of any file matching that filespec.');
        Writeln;
        Writeln('CALLING SYNTAX:');
        Writeln;
        Writeln('LOCATE <filespec>');
        Writeln;
        Writeln('For example, to find out where your screen capture files');
        Writeln('(ending in .CAP) are, you would enter:');
        Writeln;
        Writeln('LOCATE *.CAP');
        Writeln;
        Writeln('and LOCATE will show the pathname of any file ending in .CAP.');
      END
    ELSE
      BEGIN
        Writeln;
        SearchSpec := ParamStr(1);
        { A "naked" filespec searches the entire volume: }
        IF Pos('\',SearchSpec) = 0 THEN
          SearchDirectory('\',SearchSpec)
        ELSE
          BEGIN
            { This rigamarole separates the filespec from the path: }
            I := Length(SearchSpec);
            WHILE SearchSpec[I] <> '\' DO I := Pred(I);
            InitialDirectory := Copy(SearchSpec,1,I-1);
            Delete(SearchSpec,1,I);
            SearchDirectory(InitialDirectory,SearchSpec);
          END;
      END
  END.
```

The operation of **Locate** differs in small ways from the recursive search process described in connection with Figure 20.8. A single **FindFirst/FindNext** operation cannot, in fact, inspect directories for both subdirectory and file entries. To find all subdirectories, we have to use a search spec of *.*. This is due to the way DOS handles attribute bits during file searches (more on this below). To find a specific file or group of files, we need a search spec naming that file or group. Obviously, this means we must search first for directories and then, once all of a directory's child directories have been searched, inspect the files.

Therefore, **SearchDirectory** searches for subdirectories first. To make the **FindFirst** call in its search for directories, **SearchDirectory** must assemble a search spec. For directories, this is nothing more than the current path passed to **SearchDirectory** in parameter **Directory** plus the totally ambiguous file specifier *.*. Note the $10 value passed as the attribute parameter to **FindFirst**. This is an attribute byte with the subdirectory flag set to 1. The idea is that we are searching for any directory entry that has this bit set to 1.

We did something like this in searching for volume labels. We used **FindFirst** with the attribute byte set to $08, which is the value of the byte with the volume label flag set to 1. Unfortunately, DOS is inconsistent here. Executing **FindFirst** with the attribute byte set to $08 (i.e., with the volume label bit set) finds *only* the first directory entry with this bit set. Executing **FindFirst** with bit 4 (the subdirectory bit) set to 1 will find a subdirectory entry, but it will also find any ordinary file as well. A search for directory entries with the volume label bit set is *exclusive* in that it finds *only* volume label entries. A search for subdirectory entries is *inclusive* because it finds ordinary file entries *and* subdirectory entries.

This means we have to test any directory entry returned from a FIND FIRST or FIND NEXT in our search for subdirectories to make sure the returned entry is not just an ordinary file. This is done by masking out the subdirectory bit in the attribute byte and testing it:

```
IF (CurrentDTA.Attr AND $10) = $10
```

The Boolean expression here will return true only if the directory entry in **CurrentDTA** is a subdirectory entry.

Yet, as life is wont, there is another catch. All subdirectories *except* the root directory contain two additional directory entries that are not really subdirectories. You've seen these on DOS DIR displays: "." (current directory) and ".." (parent directory). Both these directory entries have bit 4 set to 1, so our search will find them. They are not part of **SearchDirectory**'s search strategy, however. We are already searching the current directory, so making a recursive call to search "." is meaningless. Furthermore, due to the way **SearchDirectory** operates, we have already searched the parent directory by the time we are searching any child directory, so searching ".." is redundant.

This is why we must also discriminate against "." and ".." in testing for subdirectory entries. So if the following Boolean expression turns up a value of **TRUE**, we know we have a subdirectory in hand:

```
IF ((CurrentDTA.Attr AND $10) = $10
AND (CurrentDTA.FileName[1] <> '.') THEN
```

This test is made within a **WHILE** loop that inherits the initialized DTA from **FindFirst** and makes calls to **FindNext** continuously until an error message indicates no more files are to be found.

Once we know we have a subdirectory entry in the DTA, we have to search it. The only tricky thing about making the recursive call is passing the correct directory path to the next instantiation of **SearchDirectory**. A string variable, **NextDirectory**, is provided to hold the string carrying the directory path to be searched by the recursive call. **NextDirectory** is loaded with the path of the directory currently being searched, in a variable called **Directory**. Generating the new path involves appending the name of the subdirectory to the end of the directory path we are currently searching. A backslash needs to be added as a separator *unless* we're still searching the root directory, whose path *is* a backslash:

```
IF Directory <> '\' THEN NextDirectory := Directory + '\'
  ELSE NextDirectory := Directory;
```

Then, after a backslash acting as a separator, the name of the *found* directory (in **CurrentDTA**) needs to be appended to **NextDirectory**:

```
NextDirectory := NextDirectory + CurrentDTA.Name;
```

Once the new path has been created in string variable **NextDirectory**, we're ready to make the recursive call. The search spec in **SearchSpec** (i.e., the file name or ambiguous file name we're searching for) is unchanged, and the directory path in **NextDirectory** reflects the name of the subdirectory we found in executing **FindFirst** and **FindNext**.

If **SearchDirectory** did not have to be called recursively (i.e., if all we had were "flat" directories, as in DOS 1.X), none of this complication would be necessary; we could simply set the DTA address when the program began running and then forget about it.

But we can't forget it. While being searched, each directory demands exclusive use of a DTA. Each time **SearchDirectory** is instantiated, a local variable called **CurrentDTA** is allocated on the stack. **CurrentDTA** is meant to function as the DOS DTA for as long as its instantiation of **SearchDirectory** has control. When an instantiation of **SearchDirectory** takes control and calls **FindFirst**, **FindFirst** sets the address of the DOS DTA to its own copy of **CurrentDTA**. When the current instantiation of **SearchDirectory** terminates and returns control to the prior instantiation of **SearchDirectory**, *that* instantiation must *again* set the address of the DOS DTA to its own copy of **CurrentDTA**. This is done within the **FindNext** call, which always sets the DOS DTA address to its own **SearchRec** parameter when it is invoked.

The first scan of a directory takes action only on subdirectories; files are ignored. Once a given scan determines that no further subdirectories exist within the current directory, a second scan is begun, this time to search for the file or files specified in string variable **SearchSpec**.

As with the search for directories, the current directory string must be combined with the search spec to produce the full path name of the file or files to be searched for. In setting up **FindFirst** to look for subdirectories, we used a search spec of *.*. This won't do here; **SearchSpec** contains the very specific file name we're trying to match.

The attribute byte is set to $00 for this search—in other words, no attribute bits are set. An attribute of $00 causes **FindFirst** to return *only* ordinary files. It will not locate directories, hidden files, system files, or volume labels. This means that any file turned up as a match is indeed a match, and no further testing (as was necessary in looking for subdirectories) of the found files needs to be done.

If **DOSError** returns error code 2 or 18, then the directory is empty and that particular instantiation of **SearchDirectory** has done its job. It terminates, either ending the **Locate** program completely or passing control back upward to the prior instantiation of **SearchDirectory** to continue the search.

More likely, a file will be discovered. Its information is translated and moved from **CurrentDTA** to a more compliant form embodied by the **DIRRec** record variable **CurrentDIR**. This translation is accomplished by the **DTAToDIR** procedure described a little earlier in this chapter. The information in **CurrentDIR** is then displayed along with the full pathname of the found file by rearranging the information generated by the **DIRToString** procedure also described earlier.

Once any information turned up by **FindFirst** is displayed, control enters a **REPEAT** loop that calls **FindNext** repeatedly, displaying any matching files it finds, until DOS error 2 or 18 turns up in **DOSError**. Either error indicates that no more matching files remain in the current directory. At that point, **SearchDirectory** terminates, either ending **Locate** or passing control back to the prior instantiation of **SearchDirectory**.

That is about all there is to **Locate**. A typical session looks like this:

```
D:\>LOCATE *.BAK

   1101  11/06/86   7:26p   \TEXT\HEADER.BAK
  11503  01/14/87  11:15a   \TURBO\LOCATE.BAK
   2623  01/13/87   7:53p   \TURBO\DIRSTRIN.BAK
   6055  01/14/87   9:33a   \TURBO\SPACER.BAK
   8595  01/14/87   9:31a   \TURBO\GETDIR.BAK
   2701  01/13/87  11:18a   \TURBO\DTATODIR.BAK
    215  12/12/86   1:12p   \CONFIG.BAK
```

It's interesting to note that the only file found in the root directory (CONFIG.BAK) turns up *last* in the display, even though DOS DIR lists it *before* both subdirectories \TURBO and \TEXT. Given your understanding of the workings of **Locate**, can you explain why this happens?

If you can, you are more than capable of writing and using recursive procedures like **SearchDirectory** that turn on their own particular brand of black magic.

20.10: SPAWNING CHILD PROCESSES WITH EXEC

DOS contains machinery to do something truly remarkable: *suspend* the currently running program and *spawn* a new program above the old program in memory. The new program behaves just as it would if you had executed it from the DOS prompt, *except* that it has less memory to work with. The program that spawned it is lurking beneath it, sleeping but still taking up room.

The running copy of the program that does the spawning is called the parent process, and the spawned program is called the child process. We call them processes because they might well be the same program, loaded from the same copy stored on disk. The Turbo Pascal Environment has this ability, and you might try executing a second copy of Turbo Pascal from the **OS** Shell item in the **F**iles menu.

There is a DOS call, EXEC, which actually accomplishes the loading and executing of child processes. Setting up registers in preparation for a call to EXEC is far from trivial, however, and it is best to think of the **Exec** procedure in the **DOS** unit as a proven black box.

The declaration for **Exec** is disarmingly simple for something so powerful:

```
PROCEDURE Exec(Path,CmdLine : String);
```

To use **Exec**, you pass the full path and file name of the program to be executed in **Path**, and any command line you wish to pass the program in **CmdLine**. Any program that may be executed from the DOS prompt may be executed from **Exec**, and any legal command line may be passed in **CmdLine**:

```
Exec('C:\UTILS\LOCATE.EXE','*.PAS');
```

This invocation of **Exec** runs the **Locate** program described in the previous section, and passes it the command line **'*.PAS'**. This is equivalent to the following DOS command line:

```
C:\>UTILS\LOCATE *.PAS
```

Too easy? Did I hear you you say that there must be a catch? There is indeed, and it involves that nastiest of all DOS bugaboos: memory management. The problem comes down to this: For a program to use **Exec** to spawn a child process, there must be enough free memory above the parent process to contain the child process. This includes the child process's data, stack, and heap.

So, how much free memory ordinarily exists above a loaded and running Turbo Pascal program? *None.*

Unless you specifically set up memory to allow it, spawning child processes will not work.

You have the power to allocate differing amounts of memory to your programs' stack and heap. The default values (i.e., those that are in force until you change them) give 16K of memory to the stack, and *all the rest of memory* to the heap. Obviously, if all remaining memory goes to the heap of the parent process, there will be no space left for spawning children.

Allocating Memory for Child Processes with $M

To work with **Exec**, then, you must rearrange memory. There are two ways to do this: One is from the menu in the Turbo Pascal Environment. The other way is through the

$M compiler command, embedded in a comment at the beginning of your program source code file for both parent and child processes.

The correct way to do it is to use the **$M** command, since memory allocation is generally something you perform on a program-by-program basis. A memory setup for a parent process would not be the same as one for a program that does not use **Exec**.

A $M command looks like this:

```
{$M 16384,0,655360}
```

The three parameters (from left to right) specify stack size, minimum heap size, and maximum heap size. This particular **$M** command is the default memory allocation for a Turbo Pascal 4.0 program. In other words, if you do not add a specific **$M** command of your own to a program, the values shown above are the ones that are in force.

The stack size figure is quite simply the amount of memory devoted to the stack segment. If you write a program with a great deal of recursion (which uses the stack to hold recursive instantiations of subprograms), you may need to increase this figure.

The minimum heap and maximum heap values will take some explanation. The amount of memory available when you execute a program is not always the same. A transient program is always the highest thing running in memory. Beneath it are DOS, DOS's device drivers, and resident programs like Sidekick, WindowDOS, READY! and so on. These things all take up memory, and your own program, running up at the top of the heap, gets what is left over. If you need more memory, you can remove resident programs from memory and perhaps do without special purpose DOS device drivers; but even then you have DOS, different versions of which are different sizes.

This forces DOS to be careful as to what it allows you to run from the command line. Baked into the beginning of any .EXE file is information that tells DOS how much room a program requires to run. Before DOS will load an .EXE file from disk and run it, it checks to see how much room the program requires, and compares that against the amount of free memory it currently has. If the program to be run needs more memory than DOS has, DOS will refuse to run it, and you'll see this error message:

```
Program too big to fit in memory
```

The minimum heap size value from the **$M** command figures into this process. The code size (which is fixed for any given sequence of program statements), stack size, and minimum heap size are what Turbo Pascal adds up to determine how much memory a program *must* have to run. This is *not* reflected in the size of the code and typed constants. No space is reserved inside a .EXE file for stack or heap.

How large you should make the minimum heap figure depends on how much heap space your program *must* have to be able to perform its work at all. If a program doesn't use the heap at all, set *both* heap figures to 0:

```
{$M 16384,0,0}
```

There's no point in specifying a maximum heap size if the minimum heap size is 0. If your program builds data structures on the heap, you'll need to make some calculations and decide on a minimum figure. Remember that if Turbo Pascal doesn't allocate any memory for the heap when compiling your program, using the heap in a program running in a system with little free memory may prevent your program from running.

It will not necessarily crash your system, but if you've loaded up a lot of TSR's and your Turbo Pascal program nearly fills the remainder of memory, trying to allocate heap space may cause your program to abort prematurely for lack of memory. Furthermore, if your program is small and there is a lot of free memory in your system, your program may run correctly even if it doesn't allocate any room for the heap. Ensuring that your programs will run under all circumstances requires keeping track of what resources they need and allocating them with **$M**.

The maximum heap figure is the amount of memory that the Turbo Pascal runtime watchdogs will allow the heap to take, if memory is available. This "permission" will be reflected in the returned values from the **MemAvail** function. If you set the maximum heap value to 655360 (essentially, all of memory), then your heap will be allowed to take all free memory DOS has available, right up to the top. You can't actually *get* 655,360 bytes for the heap (both DOS and your program's code take up some portion of that space!) but whatever is free will be available to your heap. In short, the minimum heap size figure is the amount of heap your program *needs* to run; while the maximum heap size figure is how much heap space you'd like your program to have.

Of course, if you request all available memory for your heap, DOS will grant whatever is left and then insist that no more is available for child processes. This is why, to execute a child process with **Exec**, you must set the maximum heap size figure down far enough so that there will be enough room to run the child process. The best way to do this is to make the minimum and maximum heap size allocations the same:

```
{$M 16384,32768,32768}
```

This **$M** command allocates 16K to the stack and 32K to the heap. The program will be given 32K for the heap and no more, but if it runs at all it will have *at least* 32K of heap.

The Two Kinds of Exec Errors

It may not always be possible to execute a child process, even if you have carefully kept heap size to a minimum. There may simply not be enough room after DOS and the TSRs have taken their share. There is no convenient way you can test for sufficient memory before you try to execute a child process. On the other hand, you can simply make the attempt, and if insufficient memory is available, DOS will return an error message and nothing ugly will happen. The error message is returned in the predefined variable **DOSError**. It will be one of the errors described in the table of DOS error messages on page 369 in this section, but typically you will see Error 8, Insufficient

Memory. If some other error occurs, something else (probably serious) is wrong with your program or with DOS's memory allocation system.

If there is enough memory and DOS executes the child process, there is the possibility that the child process will get in trouble somehow. If your child process aborts with some sort of DOS error, the error code will be returned in another predefined variable, **DOSExitCode**. If your parent program depends on something the child process does, you must check **DOSExitCode** after the child process returns control to the parent. It is your only indication that things went well. As with most error codes, a 0 indicates success, and anything else indicates an error of some kind. The error codes are the same ones returned by runtime errors through DOS ERRORLEVEL.

Executing DOS Commands through COMMAND.COM

One of the most interesting programs to run as a child process is COMMAND.COM, the doorway to DOS itself. COMMAND.COM has some secret hooks into DOS, certainly, but by and large it is only a command parser that contains some very limited command code of its own. DIR, ERASE, and COPY are its most commonly-used built-in commands, and you can execute them from within a program by spawning COMMAND.COM as a child process.

It isn't enough just to execute COMMAND.COM. You need to pass it the name of the command you wish to execute, and that is done through the command line string passed along with the name of the program to be executed. For example, to execute a DIR command from within a Turbo Pascal program, you would use this syntax:

```
Exec('C:\COMMAND.COM','/C DIR *.PAS');
```

You must be sure to pass the full path to COMMAND.COM if it is not in the current directory. The /C is a special directive indicating to COMMAND.COM that a built-in command is present on the same command line.

Keep in mind that you *only* have to execute COMMAND.COM to invoke a built-in command; that is, a command that is not itself a separate program like FORMAT.COM or CHKDSK.COM.

"Shelling Out" to DOS

Ducking out of your program to the DOS prompt temporarily is one of the most powerful uses of **Exec**. To pull this trick, all you need to do is execute **COMMAND.COM** by itself without a command line:

```
Exec(C:\COMMAND.COM,'');
```

That's all there is to it; well, sort of. You will see the COMMAND.COM signon message, and then the DOS prompt will appear:

```
The IBM personal Computer DOS
Version 3.10 (C)Copyright International Business Machines 1981, 1985
             (C)Copyright Microsoft Corp 1981, 1985

C:\>
```

You can then do anything you could do at the DOS prompt under ordinary circumstances, within the possible limitation of having a smaller available memory pool to draw on. When you wish to return to the parent program, simply type EXIT at the command prompt. At that point the copy of COMMAND.COM that had been running as a child process will terminate, and you will return to your parent program.

One thing to remember is that nothing magical is done to preserve the screen as it exists at the moment you "shell out" to DOS. If you disrupt the screen while working from the DOS prompt, the screen will remain disrupted when you return to the parent program. Neither will the parent program's cursor position be retained. To bring back the parent program's screen as it was when you left it, you must save that screen somewhere along with the cursor position, and restore them when you return.

The obvious place to store a screen pattern is on the heap, where space is relatively plentiful. Moving information to and from the heap can be done very quickly with the **Move** statement (see Section 23.4) for the sizes we're talking about, between 4000 and 8000 bytes.

The procedure **SaveScreen** moves the current screen pattern onto the heap, and the procedure **RestoreScreen** moves it back from the heap into display memory. Keeping a screen intact across a DOS command session becomes no more difficult than this:

```
SaveScreen(ScreenPointer);
Exec('C:\COMMAND.COM','');
RestoreScreen(ScreenPointer);
```

Here, **ScreenPointer** is a generic pointer, and the routines are interesting because they utterly hide the details of how and where the saved screen is stored.

```
{->>>>SaveScreen<<<<--------------------------------------------}
{                                                               }
{ Filename : SAVESCRN.SRC -- Last Modified 7/14/88              }
{                                                               }
{ This routine saves the current display buffer out to the      }
{ heap, regardless of how large the current display format is.  }
{ The routine queries the BIOS to determine how many lines are  }
{ currently on the screen, and saves only as much data to the   }
{ heap as necessary.                                            }
{                                                               }
{ Use the companion routine, RestoreScreen, to bring a screen  }
{ back from the heap.                                           }
{                                                               }
{                                                               }
{                                                               }
{---------------------------------------------------------------}

PROCEDURE SaveScreen(VAR StashPtr : Pointer);

TYPE
  VidPtr   = ^VidSaver;
  VidSaver = RECORD
               Base,Size : Word;
               BufStart  : Byte
             END;

VAR
  VidBuffer : Pointer;
  Adapter   : AdapterType;
  StashBuf  : VidSaver;
  VidVector : VidPtr;

BEGIN
  Adapter := QueryAdapterType;
  WITH StashBuf DO
    BEGIN
      CASE Adapter OF
        MDA,EGAMono,VGAMono,MCGAMono : Base := $B000;
        ELSE Base := $B800;
      END;  { CASE }
      CASE DeterminePoints OF
        8  : CASE Adapter OF
               CGA                 : Size := 4000;  { 25-line screen }
               EGAMono,EGAColor : Size := 6880;  { 43-line screen }
               ELSE                  Size := 8000;  { 50-line screen }
             END; { CASE }
        14 : CASE Adapter OF
               EGAMono,EGAColor : Size := 4000;  { 25-line screen }
               ELSE                  Size := 4320;  { 27-line screen }
             END; { CASE }
        16 : Size := 4000;
      END; { CASE }
      VidBuffer := Ptr(Base,0);
    END;

  GetMem(StashPtr,StashBuf.Size+4);  { Allocate heap for whole shebang }
  { Here we move *ONLY* the VidSaver record (5 bytes) to the heap: }
  Move(StashBuf,StashPtr^,Sizeof(StashBuf));
  { This casts StashPtr, a generic pointer, to a pointer to a VidSaver: }
  VidVector := StashPtr;
  { Now we move the video buffer itself to the heap.  The vide data is }
  { written starting at the BufStart byte in the VidSaver record, and  }
  { goes on for Size bytes to fit the whole buffer.  Messy but hey, this }
  { is PC land! }
  Move(VidBuffer^,VidVector^.BufStart,StashBuf.Size);
END;
```

```
{->>>>RestoreScreen<<<<---------------------------------------}
{                                                              }
{ Filename : RSTRSCRN.SRC -- Last Modified 7/14/88             }
{                                                              }
{ This routine accepts a pointer generated by the SaveScreen   }
{ routine and brings back the saved display buffer from the    }
{ heap and loads it into the refresh buffer of the display     }
{ card.                                                        }
{                                                              }
{                                                              }
{                                                              }
{--------------------------------------------------------------}

PROCEDURE RestoreScreen(StashPtr : Pointer);

TYPE
  VidPtr    = ^VidSaver;
  VidSaver = RECORD
               Base,Size : Word;
               BufStart  : Byte
             END;

VAR
  VidVector : VidPtr;
  VidBuffer : Pointer;
  DataSize  : Word;

BEGIN
  VidVector := StashPtr;  { Cast generic pointer onto VidSaver pointer }
  DataSize  := VidVector^.Size;
  { Create a pointer to the base of the video buffer: }
  VidBuffer := Ptr(VidVector^.Base,0);
  { Move the buffer portion of the data on the heap to the video buffer: }
  Move(VidVector^.BufStart,VidBuffer^,VidVector^.Size);
  FreeMem(StashPtr,DataSize + 4);
END;
```

Most of the trickiness of the system lies in **SaveScreen**, and that stems from the fact that a screen may be 25, 27, 43, or 50 lines in size. The routine determines the size of the screen by using the **DeterminePoints** function described in Section 18.4. The size of the screen is determined by the size of the font currently in use. Another more familiar complication is that video display buffers in the PC family may exist in one of two places: at segment $B000 (for monochrome modes) or $B800 (for color modes). This is determined by using the **QueryAdapterType** function, also discussed in Section 18.4.

Once the size and location of the current screen have been determined, the rest is a suite of tricks with pointers. A generic pointer must be passed to **SaveScreen**, and this pointer is used to reference a block of storage on the heap allocated with **GetMem**. **GetMem** allocates enough space on the heap to hold the screen itself plus four additional bytes. These four bytes provide room for two **Word** variables indicating the size of the screen and the segment base of the video buffer.

The data are moved to the heap in two operations. The first moves only the segment base value and the screen size, in the form of a record type called **VidSaver**. There is a "dummy" field in **VidSaver**, called **BufStart**, that exists only to provide a starting point for the second move operation. This second move operation moves the video buffer data from the video buffer to the heap address at which **BufStart** exists.

The **RestoreScreen** routine is considerably simpler. It receives the pointer to the stored screen on the heap, and casts that pointer onto **VidVector**, a pointer to the **VidSaver** type. This allows **RestoreScreen** to easily bring the video buffer segment and screen size values from the heap by dereferencing the **VidVector** pointer. With the screen size and buffer segment in hand, moving the screen data from the heap to the display buffer is nothing more than another call to Turbo Pascal's **Move** procedure.

The only caution in shelling out to DOS between calls to **SaveScreen** and **RestoreScreen** is this: Don't change the size of the screen while you're out working with DOS. **RestoreScreen** assumes when it goes to work that the size of the screen is what it was when **SaveScreen** moved the screen image onto the heap. If the screen size is different, you won't see what you were seeing when you ducked out to COMMAND.COM.

The short demo program **ShellOut** shows **SaveScreen** and **RestoreScreen** in use. A simple screen is constructed, then saved out to the heap just before executing COMMAND.COM. Once COMMAND.COM returns, **RestoreScreen** brings the screen back in from the heap, correcting any disruption that occurred during the interactive session with COMMAND.COM.

```
{--------------------------------------------------------------}
{                            ShellOut                          }
{                                                              }
{     "Shell to DOS" with screen save demonstration program    }
{                                                              }
{                              by Jeff Duntemann               }
{                              Turbo Pascal V4.0               }
{                              Last update 7/1/88              }
{                                                              }
{                                                              }
{                                                              }
{--------------------------------------------------------------}

PROGRAM ShellOut;

{$M 16384,16384,16384}  { Reserve some heap to save the screen }

USES DOS,CRT,BoxStuff;  { Boxstuff described in Section 17.3 }

TYPE
  AdapterType = (None,MDA,CGA,EGAMono,EGAColor,VGAMono,
                 VGAColor,MCGAMono,MCGAColor);

VAR
  Stash : Pointer;


{$I WRITEAT.SRC}         { Described in Section 18.3 }
{$I QUERYDSP.SRC}        { Described in Section 18.4 }
```

```
{$I FONTSIZE.SRC}       { Described in Section 18.4 }


PROCEDURE SaveScreen(VAR StashPtr : Pointer);

TYPE
  VidPtr   = ^VidSaver;
  VidSaver = RECORD
               Base,Size : Word;
               BufStart  : Byte
             END;

VAR
  VidBuffer : Pointer;
  Adapter   : AdapterType;
  StashBuf  : VidSaver;
  VidVector : VidPtr;

BEGIN
  Adapter := QueryAdapterType;
  WITH StashBuf DO
    BEGIN
      CASE Adapter OF
        MDA,EGAMono,VGAMono,MCGAMono : Base := $B000;
        ELSE Base := $B800;
      END;  { CASE }
      CASE DeterminePoints OF
        8  : CASE Adapter OF
               CGA               : Size := 4000;  { 25-line screen }
               EGAMono,EGAColor : Size := 6880;  { 43-line screen }
               ELSE               Size := 8000;  { 50-line screen }
             END; { CASE }
        14 : CASE Adapter OF
               EGAMono,EGAColor : Size := 4000;  { 25-line screen }
               ELSE               Size := 4320;  { 27-line screen }
             END; { CASE }
        16 : Size := 4000;
      END; { CASE }
      VidBuffer := Ptr(Base,0);
    END;

  GetMem(StashPtr,StashBuf.Size+4);  { Allocate heap for whole shebang }
  { Here we move *ONLY* the VidSaver record (5 bytes) to the heap: }
  Move(StashBuf,StashPtr^,Sizeof(StashBuf));
  { This casts StashPtr, a generic pointer, to a pointer to a VidSaver: }
  VidVector := StashPtr;
  { Now we move the video buffer itself to the heap.  The vide data is }
  { written starting at the BufStart byte in the VidSaver record, and  }
  { goes on for Size bytes to fit the whole buffer.  Messy but hey, this }
  { is PC land! }
  Move(VidBuffer^,VidVector^.BufStart,StashBuf.Size);
END;



PROCEDURE RestoreScreen(StashPtr : Pointer);

TYPE
```

```
  VidPtr   = ^VidSaver;
  VidSaver = RECORD
               Base,Size : Word;
               BufStart  : Byte
             END;

VAR
  VidVector : VidPtr;
  VidBuffer : Pointer;
  DataSize  : Word;

BEGIN
  VidVector := StashPtr;  { Cast generic pointer onto VidSaver pointer }
  DataSize  := VidVector^.Size;
  { Create a pointer to the base of the video buffer: }
  VidBuffer := Ptr(VidVector^.Base,0);
  { Move the buffer portion of the data on the heap to the video buffer: }
  Move(VidVector^.BufStart,VidBuffer^,VidVector^.Size);
  FreeMem(StashPtr,DataSize + 4);
END;


BEGIN
  ClrScr;
  MakeBox(1,1,80,24,GrafChars);
  WriteAt(-1,3,True,False,'** Relatively Dumb Parent Program **');
  GotoXY(10,10); Write('Press CR to shell to DOS: '); Readln;
  SaveScreen(Stash);
  Exec('C:\COMMAND.COM','');
  RestoreScreen(Stash);
END.
```

20.11: READING THE DOS ENVIRONMENT (VERSION 5.0)

One of the little strangenesses of PC DOS V2 and later is an item called the *DOS environment block* (usually referred to as "*the environment*"). What was at the outset a good idea was implemented so peculiarly that it leads me to wonder if the lights are on with nobody home up in Washington State.

The underlying idea is this: DOS reserves a region of memory somewhere, in a location that doesn't change, as a "bulletin board" reserved for the posting of string-data notices. These notices are for applications programs that load, run for awhile under DOS, and then terminate. They have the ability to peek at the bulletin board and see what notices have been posted.

The notices can be anything at all, but are typically useful things about the DOS configuration of the system: Where COMMAND.COM is stored, what the current DOS search path is for program execution, and so on. Additionally, the user has the ability to place notices in the DOS environment by using DOS's SET command. By entering a command like:

```
SET SYMLIB=D:\M2LIB\SYM
```

it's possible to post the string "SYMLIB=D:\M2LIB\SYM" on the DOS environment's bulletin board. Then, when a utility program needs to know where the Modula 2 symbol files are kept, it can peek at the environment and get the path attached to the identifier "SYMLIB."

So far so good. This solves the knotty problem of sharing critical information among sequentially transient but related programs. Now, however, for the catch: DOS reserves only 160 bytes of space for its environment block. The very latest in DOS versions (3.2) allows this size to be increased to a maximum of 32K, but to my mind the additional features of DOS 3.2 don't warrant another $90 out of my pocket.

For the vast majority of DOS users, 160 bytes is all you get. That's room for a typical 80-byte search path, COMSPEC, and one, *maybe* two, other things. It does *not* provide room for a search path, COMSPEC, and the five or ten environment variables requested by many language systems like Logitech's excellent Modula 2/86. Every time I need to use Logitech Modula 2, I must reboot my machine with a special batch file that eliminates my search path to make room for the five Modula 2 environment variables.

In short, for DOS versions prior to 3.2, the DOS environment is close to useless. Nonetheless, if you're the forward-looking type, you might wish to make use of the environment in your own programs. The **DOS** unit for Turbo Pascal 5.0 provides several routines for reading the environment.

The Nature and Location of the DOS Environment

The environment is an area of reserved memory within DOS in which strings may be written. The strings stored in the environment are ASCIIZ strings, meaning that they are arrays of characters terminated with a binary 0 (in Pascal, **Chr(0)**). The end of significant information in the environment is signaled by the presence of another binary 0 after the binary 0 that signals the end of the last string in the environment. If the environment *begins* with a binary 0, nothing has been stored into it.

The location of the DOS environment is stored at the beginning of the code segment of your Pascal program, in a 256-byte area called the "Program Segment Prefix," or PSP. At offset $2C into the PSP is a 2-byte segment address, which is the segment of the first byte of the DOS environment. (The offset of the first byte of the environment is always assumed to be 0.) Therefore, finding the environment is only as difficult as saying:

```
EnvSegment := MEMW[CSEG : $2C];
```

where **EnvSegment** is type **Word**.

You don't have to worry about where the environment is in memory, however; Turbo Pascal locates it without assistance.

Counting Environment Strings (Version 5.0)

If you're interested in reading everything the environment contains, it's helpful to know how many strings have been stored there. Turbo Pascal 5.0's **EnvCount** function returns the number of individual strings existing in the environment. **EnvCount** is declared this way:

```
FUNCTION EnvCount : Integer;
```

Fetching Environment Strings by Index (Version 5.0)

Once you know how many strings are lurking in the environment, retrieving them is done with the **EnvStr** function. **EnvStr** is passed the index of a string in the environment. The first string is string number 1, the second string number 2, and so on up to the number of strings reported by **EnvCount** as explained above. **EnvStr** is declared this way:

```
FUNCTION EnvStr(Index : Integer) : String;
```

A one-statement program can display the full contents of the DOS environment to your screen, using **EnvCount** and **EnvStr**:

```
PROGRAM ShowEnvironment;

USES DOS;

VAR
  I : Integer;

BEGIN
  IF EnvCount > 0 THEN
    FOR I := 1 TO EnvCount DO
      Writeln(EnvStr(I))
END.
```

Reading a Named Environment String (Version 5.0)

If you know the name of an environment string (i.e., the portion of a string on the left side of the = sign), Turbo Pascal provides a way of reading the portion of the string on the right side of the = sign by using its **GetEnv** function. **GetEnv** is declared this way:

```
FUNCTION GetEnv(EnvironmentVariable : String) : String;
```

The use of **GetEnv** is best explained by example. The following short program prompts for an environment variable name (such as PATH, or COMSPEC) and searches the environment for a string of that name. If the string is found, the program displays its value:

```
PROGRAM ShowEnvironmentVariable;

USES DOS;

VAR
  Name,Value : String;

BEGIN
  Write('Enter the name of an environment variable: ');
  Readln(Name);
  Value := GetEnv(Name);
  IF Length(Value) = 0 THEN
    Writeln('That variable is not in the environment.')
  ELSE
    Writeln('The value of ',Name,' is ',Value,'.');
END.
```

For example, you might see the following session:

```
Enter the name of an environment variable: PATH
The value of PATH is C:\DOS;D:\UTILS;D:\WP42;C:\TURBO5
```

If **GetEnv** cannot find the named string, it returns a 0-length string.

21

Dynamic Variables and Pointers

21.1: MANAGING THE HEAP

Back in Chapter 10 we described dynamic variables and pointers in some detail, but we said very little about where dynamic variables actually exist. Like everything else in a computer program, they exist in electronic memory. What is important is how they are stored there and how remaining storage space is managed.

We have used the terms heap and heapspace informally without defining them. *Heapspace* is that region of memory set aside for allocation to dynamic variables. When you say:

```
New(APtr);
```

you have created a brand new dynamic variable that did not exist before. It exists in heapspace, and it is accessed through the pointer variable **APtr**.

When a program begins to run, it reserves some quantity of RAM for dynamic variables. In the Turbo Pascal environment, the default condition is that all memory not allocated to code, static data, and stack will be allocated to the heap. This parceling out of memory can be modified by using the **$M** compiler command, as explained in Section 27.3. **$M** is also discussed in Section 20.10, in connection with the **Exec** procedure in Turbo Pascal's standard **DOS** unit.

MemAvail and MaxAvail

Knowing how much memory is actually available for dynamic variables can be critical. The Turbo Pascal runtime code keeps track of how much memory remains for the heap, and if you request more than is available, a runtime error will be generated.

The key is not to create a dynamic variable that is larger than the available heap memory. Turbo Pascal provides two functions for measuring available memory: **MemAvail** and **MaxAvail**.
They are predeclared:

```
FUNCTION MemAvail : LongInt;

FUNCTION MaxAvail : LongInt;
```

MemAvail returns a value that indicates, in bytes, the total amount of memory available for dynamic variables. Note that this is different from **MemAvail** for Turbo Pascal 3.0, which returned the value in *paragraphs* to avoid overflowing a 16-bit integer. Long integers have put that problem into the past; now, when you ask for the amount of free heap space, there is nothing further to calculate—what **MemAvail** tells you is what you have.

MaxAvail returns a value, again in bytes, that is the size of the largest single block of free memory in the heap.

The difference between **MemAvail** and **MaxAvail** has to do with how the heap works. Dynamic variables are created with **New** (see Section 10.3). When you're finished with a dynamic variable, you can erase it with **Dispose** (see Section 10.5). **Dispose** frees up the memory formerly occupied by the dynamic variable. The problem is that, unlike the stack, which holds data items in strict order and releases them in strict order, the heap holds dynamic variables anywhere it has room. Disposing of them at random tends to cut the heap up into small slices of free memory, each where a disposed dynamic variable used to exist.

This means that it is possible to have 16K of heap space according to **MemAvail**, and *still* not be able to create a new dynamic variable. If there is no single free block of memory large enough to hold a dynamic variable of a given type, you cannot safely invoke **New** and create that variable. To do so will trigger runtime error:

```
Error 203:  Heap overflow
```

and terminate your program. There are things you can do to temper this error a little, but they are advanced techniques and we'll cover them in Section 21.5.

Thus, the function of **MaxAvail** is to determine if it is possible to safely create any given dynamic variable:

```
IF MaxAvail < SizeOf(MyType)
  THEN Writeln('Not enough room!');
```

This example calculates the size that a dynamic variable will be by invoking **SizeOf** on the *type* of the dynamic variable. If that size figure is greater than the value returned by **MaxAvail**, then no single block of memory exists that is large enough to hold another dynamic variable of that type.

What can you do if you find you have no spot large enough to create a new dynamic variable? Nothing automatic; there is no facility in Turbo Pascal that can shuffle the heap around, and gather free memory together into a single contiguous block. (This is sometimes called "garbage collection," and it is *very* hard to do.) About all you can do is start disposing of existing dynamic variables until a large enough block opens up somewhere.

The real answer, of course, is to anticipate this problem and not keep large dynamic variables or structures (like linked lists or trees, see below) hanging around in the heap when they are not needed. Create them. Use them. Then *immediately* throw them away!

21.2: LINKED LISTS

The whole point of having dynamic variables is to be able to create and destroy them as needed, without having to predeclare them at compile time. For single variables, this hardly seems worth the bother. Dynamic variables really come into their own when you begin building entire structures of variables out in heapspace. These are structures whose size and shape may not be known at compile time, and which, depending on the job the program is called upon to do, may change drastically from execution to execution.

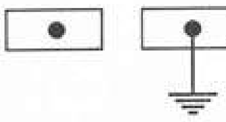
The subject of data structures is a large and difficult one. It encompasses structures like circular buffers, linked lists, doubly linked lists, and trees. Space will not allow us to go deeply into dynamic data structures; that subject can itself do justice to an entire book. As an example of what dynamic variables can do, we will look at simple linked lists. Linked lists lend themselves well toward representing in diagram form, and Figure 21.1 summarizes the different symbols I'll be using to represent pointers and the records that make up singly and doubly linked lists discussed in this section.

When we first looked at dynamic variables, back in Chapter 10, we saw that a dynamic variable is tethered to its program's static data area by a pointer variable. Dynamic variables have no names, so the only way to use them is to work from the pointer variable, which does have a name.

Figure 21.1

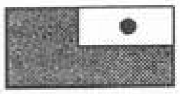
Typographical Conventions for Linked Lists

①



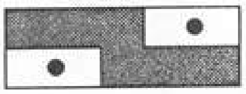
Pointers. The left pointer is undefined. The right pointer has been set to NIL.

②



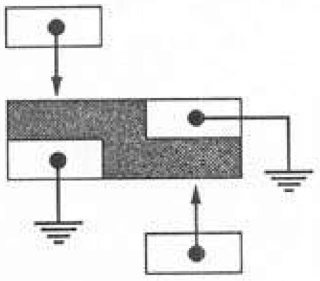
A record containing one pointer. Singly linked lists are built from this kind of record.

③



A record containing two pointers. Doubly linked lists are built from this kind of record.

④



Two pointers pointing to the same record. The pointers are considered to be equal, even though they touch the record's symbol at different spots. Both pointers within the record have been set to NIL.

But consider a record type that includes, as one of its fields, a pointer variable to another record like itself:

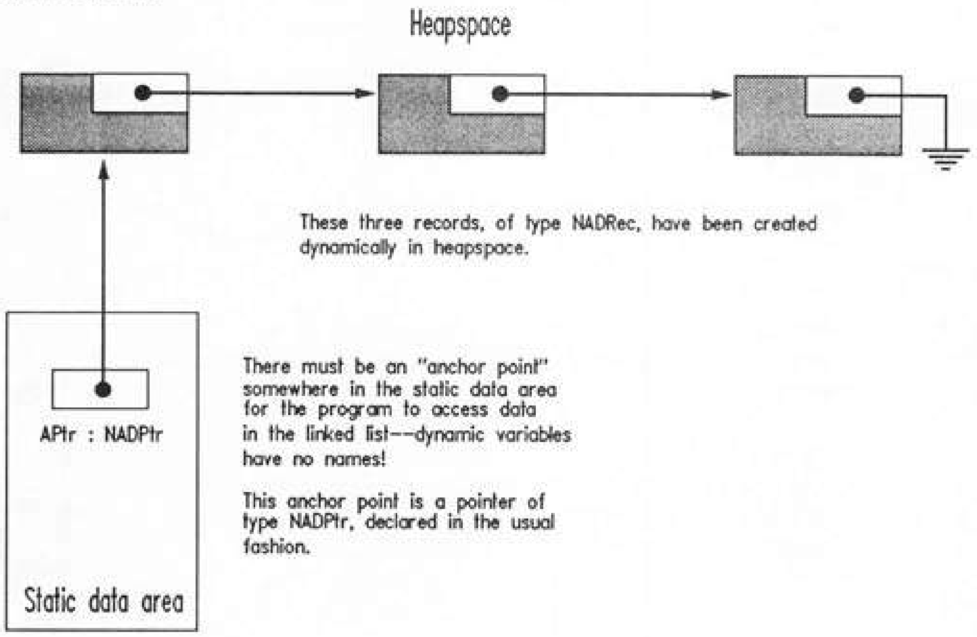
```
TYPE
  NADPtr = ^NADRec;

  NADRec = RECORD
             Name    : String;
             Address : String;
             City    : String;
             State   : String;
             Zip     : String;
             Next    : NADPtr
           END;
```

One of these records, once created out in heapspace and tethered to the static data area, could be made to point to another such record out in heapspace, and it to another, and so on. We could have an entire string of records, each one pointing to the next, with the whole tied to our program by just a single pointer variable.

Such a collection of dynamic records linked by pointers is called a *linked list* (see Figure 21.2). The very last record in the list contains a pointer whose value is **Nil**.

Figure 21.2

A Linked List


Heapspace
These three records, of type NADRec, have been created dynamically in heapspace.
APtr : NADPtr
There must be an "anchor point" somewhere in the static data area for the program to access data in the linked list--dynamic variables have no names!
This anchor point is a pointer of type NADPtr, declared in the usual fashion.
Static data area

Again, remember that **Nil** only means a pointer that points to nothing. The **Nil** pointer signals the end of the linked list.

Traversing a Linked List

As with any dynamic variable, the list has no name. Its tether to reality is a pointer variable with a name. But neither the records themselves nor the pointers that link them have names. The pointer that tethers the list to static data is often called the *root*.

How do we access the records in a linked list? The first record, fairly obviously, is **Root^**, because the pointer variable **Root** points to it. But how about the record pointed to by the pointer buried in **Root^**? You can in fact say **Root^.Next^**, which can be read, somewhat awkwardly, as "the record pointed to by the **Next** pointer contained in the record pointed to by **Root**." Obviously, it gets still worse for the third record in the list, **Root^.Next^.Next^**. Subject to certain limits of the compiler, you can go on this way for many levels: **Root^.Next .Next^.Next^.Next^**, and so on.

But that won't work. Why? *It assumes we know how many records are in the linked list at compile time.* And, you may recall, the reason to use dynamic variables is to create structures whose size and shape may *not* be known at compile time.

No, the way to deal with dynamic structures is to be dynamic. We must move along the linked list, looking at one record at a time, until we detect the end of the list. This process is called *traversing* a linked list.

Traversing a list requires a little extra machinery. Specifically, in addition to the root pointer, we need a new pointer to help us. Look over the following code that displays data from records of a linked list pointed to by **Root**:

```
Current := Root;     { Current and Root both point to list }

WHILE Current <> Nil DO
  BEGIN
    ShowRecord(Current^);  { Displays data in record Current^ }
    Current := Current^.Next   { Points Current to next record }
  END;
```

Assigning one pointer to another, you'll recall, means that both pointers now point to the same item. **Current** and **Root** both point to the first record in the list. **Root** is the the list's anchor into reality; we must *not* change it without good reason.

But we can make **Current** *travel* along the list by assigning to it the value of the **Next** field in the record it currently points to. Eventually **Current** is pointing to the last record in the list. The **Next** field inside that last record has a value of **Nil**. When **Current** is assigned **Current^.Next**, then **Current** takes on the value **Nil**. The **WHILE** loop is thus satisfied and the loop terminates.

Building Linked Lists

We covered traversing a linked list before explaining how a linked list is built, because traversing the list can be part of the process of building the list. How the list is built depends upon your needs.

Linked lists are either *ordered* or not. An ordered list is built such that when you traverse the list, you will encounter the items in the list in some sort of order according to the data carried in the items. If, as you traverse a list, you find that all the items are in alphabetical order by one of the item's data fields, then that is an ordered list.

Building a list that is not ordered is simplicity itself. Given **GetRec**, a procedure that supplies filled records (of type **NADRec** as given above), this code will construct a linked list:

```
VAR
  WorkRec      : NADRec;
  Root,Holder  : NADPtr;

Root := Nil;
Holder := Root;
FOR I := 1 TO 10 DO
  BEGIN
    GetRec(WorkRec);         { Fill the (static) work record  }
    New(Root);               { Create empty dynamic record    }
    WorkRec.Next := Holder;  { Copy root pointer into WorkRec }
    Root^ := WorkRec;        { Copy work record into Root^    }
    Holder := Root          { Point Holder to list root again }
  END;
```

The critical point here is that new records are inserted at the *beginning* of the list rather than at the end. The very first record loaded into the list becomes the "tail" of the list, and is pushed further from the root by each new record returned by **GetRec**.

A short list may not have to be ordered, because even a beginning-to-end sequential search on a short list (less than 100 to 200 items) is so fast that ordering it is not worth the small amount of time saved in searches.

But there are other reasons to order a list: We may want to *use* the data in some order in addition to just searching it for matching data. For example, searching for names in an unordered list may be acceptably fast, but you wouldn't want to print out the names (say, for an address book) in random order.

Building an ordered list involves some traversing. Briefly, to add an item to an ordered list you must traverse the list until you find the place in the list where the new item must be inserted to keep the list in order. Once you find the proper spot, you insert the item. The following procedure builds an ordered list of records by adding a record in proper order to a list pointed to by **Root**:

```
PROCEDURE Builder(VAR Root : NADPtr; WorkRec : NADRec);

VAR
  Current,Last : NADPtr;

BEGIN
  IF Root = Nil THEN        { List is empty! }
    BEGIN
      New(Root);
      WorkRec.Next := Nil;
      Root^ := WorkRec
    END
  ELSE                      { List already contains some records }
    BEGIN
      Current := Root;
      REPEAT                { Traverse list to find correct spot }
        Last := Current;
        Current := Current^.Next
      UNTIL (Current = Nil) OR (Current^.Name > WorkRec.Name)
                            { Found spot--now insert new record  }
      IF Root^.Name > WorkRec.Name THEN   { Record becomes new }
        BEGIN                             { first item on list }
          New(Root);                { Create new record for Root }
          WorkRec.Next := Last;     { Copy root pointer to Next  }
          Root^ := WorkRec          { Copy WorkRec to new record }
        END
      ELSE                                { Record belongs in   }
        BEGIN                             { mid-list somewhere }
          New(Last^.Next);           { Create new record for Last}
          WorkRec.Next := Current;   { Point new rec to Current   }
          Last^.Next := WorkRec;     { Copy WorkRec to new record}
        END
    END
END;
```

There are three possibilities that must be dealt with separately:

1. The list is empty. **Root** is equal to **Nil**. The record passed to **Builder** becomes the first record in the list. No traversing is necessary at all.
2. Testing the first record in the list (**Root^**) shows that the record to be inserted must be inserted *before* **Root^**. The new record must then become **Root^**, and its **Next** field must be made to point to the old **Root^**.
3. Traversing the list shows that the record must be inserted somewhere within the list or at its end. The record must be inserted between two existing records, or between the last record and **Nil**.

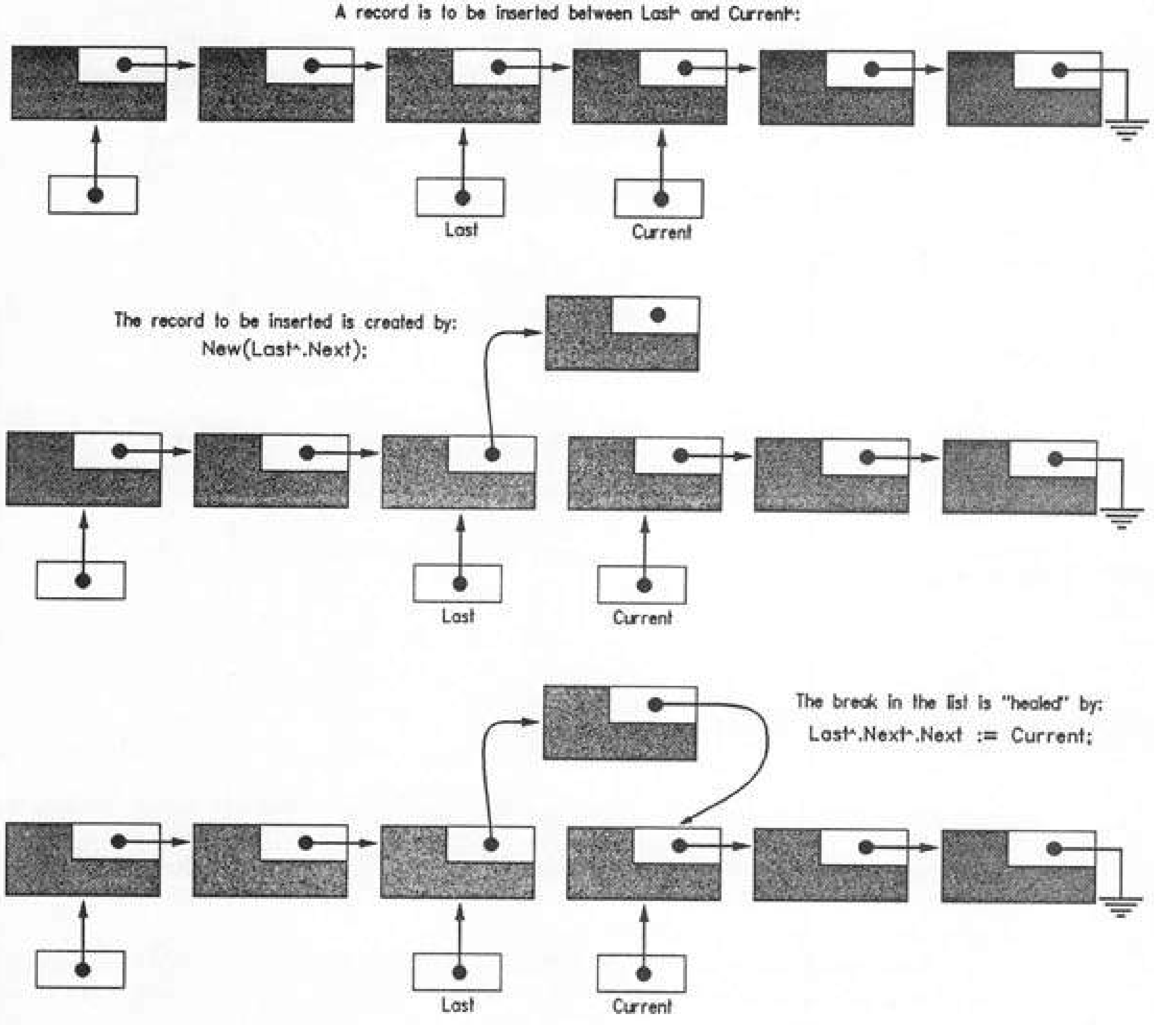
It is to cater to the third possibility that the pointer **Last** was created (Figure 21.3). **Last**, as its name implies, points to the last record pointed to by **Current**. **Last** is always one record behind **Current**, once **Current** moves off "home base." Having **Last** right behind **Current** makes it easy to insert a record into the list. With **Current** safely pointing to the rest of the list, the list is broken at **Last^.Next** and a new dynamic record is created for **Last^.Next** to point to:

```
New(Last^.Next);
```

Now to "heal" the break in the list: **WorkRec**'s **Next** field is pointed to **Current^**. **WorkRec** is now part of the second fragment of the list. Finally, **WorkRec** is assigned

Figure 21.3

Inserting a Record into a Linked List


A record is to be inserted between Last^ and Current^:
Last
Current
The record to be inserted is created by:
New(Last^.Next);
Last
Current
The break in the list is "healed" by:
Last^.Next^.Next := Current;
Last
Current

to **Last^.Next**. The list is whole once again, with **WorkRec** inserted between **Last^** and **Current^**.

Disposing of a Linked List

We have emphasized that you don't throw away a linked list by cutting the list free of its root pointer. The space in memory is still occupied, albeit by unreachable and now useless records. A linked list must be disposed of in an orderly manner. In some respects it is a mirror image of the method used to create an unordered linked list. Since the order of the records doesn't matter when you're disposing of them, the same method is used to dispose of ordered and unordered lists (Figure 21.4).

Pointer **Root** anchors the list to be disposed of. A second pointer, **Holder**, is also needed. With **Root** pointing to the first record in the list, **Holder** is made to point to the second record. Now **Holder** is anchoring the list from one record further on than **Root**. **Root^** can thus be safely disposed of.

Root is then pointed to **Holder^**. With **Root** now also at the new beginning of the list, **Holder** may be moved up to the second record in the list, allowing **Root^** to be disposed of. The loop continues until no more records remain.

```
PROCEDURE ListDispose(VAR Root : NADPtr);

VAR
  Holder : NADPtr;

BEGIN
  IF Root <> Nil THEN            { Can't dispose if no list!   }
    REPEAT
      Holder := Root^.Next;      { Grab the next record...     }
      Dispose(Root);             { ...dispose of the first...  }
      Root := Holder             { and make the next the first }
    UNTIL Root = Nil
END;
```

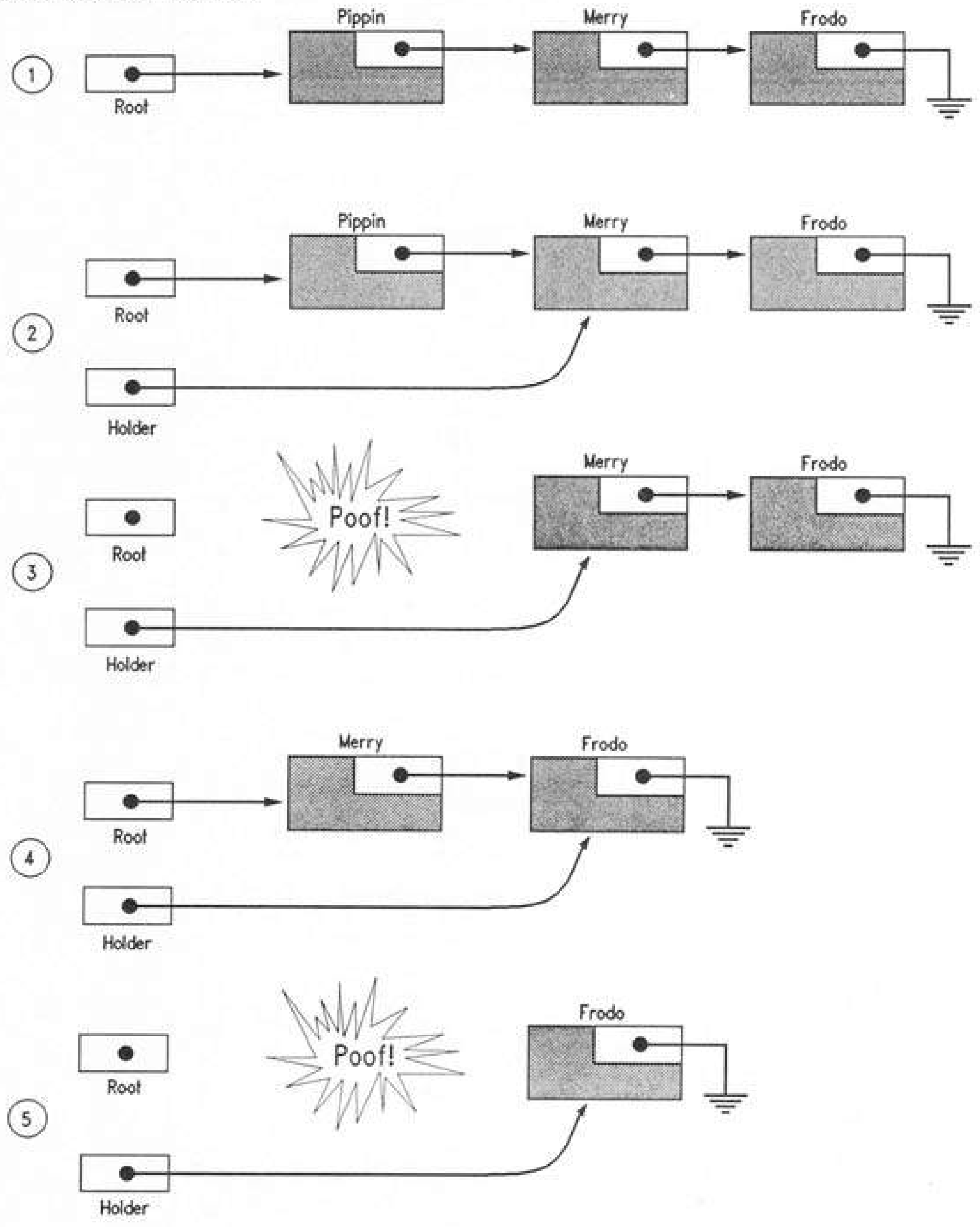
21.3: A SIMPLE LIST MANAGER PROGRAM

To reinforce all the various ways of manipulating linked lists, in this section we'll present a simple list manager program. **ListMan** is the skeleton of a "personal address book" to contain names, addresses, and (if you add another field to the **NADRec** record type) phone numbers. **ListMan** is more than just a demonstration of linked list handling. It is your chance to show your stuff by building on it.

ListMan creates and adds records to ordered lists, saves them to disk, loads them from disk, and disposes of them gracefully when they are no longer needed. After each

Figure 21.4

Disposing of a Linked List



change to the size of the list, the number of records that may be added is displayed at the top of the screen.

The **AddRecords** procedure in **ListMan** is an expansion of the **Builder** procedure given above. **AddRecords** performs an additional service: It looks for duplicate names. There may be two distinct John Browns in your Christmas card list, but you also may have accidentally typed his name in twice. **AddRecords** shows you both duplicate records on the screen at once, and allows you to throw away the most recently entered one if it is in fact a duplicate.

CheckSpace shows how **MemAvail** is used. **MaxAvail** is not used because records are not deleted at random. Lists are deleted all at once, so no heapspace fragmentation will occur. If you were to expand **ListMan** to allow the deletion of records from the list at random (a good idea; what if you don't get a Christmas card from John Brown this year?), you will have to begin using **MaxAvail**.

Another function lacking in **ListMan** is search-and-edit. In other words, if John Brown moves, you will need to search the list for his name and change the address data stored in his record. The code to do this could be cribbed from **AddRecords**, which already searches the list for duplicates of the record to be inserted. You might, in fact, "break out" the search-for-duplicate code into a separate procedure that may be called by both **AddRecords** and your hypothetical search-and-edit procedure. In the same way, you could use the **GetString** procedure given in Section 15.2 to write a more general data entry/edit procedure replacing the sequence of **Readln**s in **AddRecords**. Both **AddRecords** and your search-and-edit procedure could call the new data entry/edit procedure.

This is much of the magic of Pascal: The building of toolkit routines that may be called by two or more procedures in a program. In this way, the richness of function of a program may be increased tremendously without turning the program into something of Godzillan size.

```
{--------------------------------------------------------------}
{                          ListMan                             }
{                                                              }
{    Mailing list manager demo using dynamic (heap) storage    }
{                                                              }
{                              by Jeff Duntemann               }
{                              Turbo Pascal V5.0               }
{                              Last update 7/24/88             }
{                                                              }
{                                                              }
{                                                              }
{--------------------------------------------------------------}

PROGRAM ListMan;

USES Crt;

TYPE
  String30 = String[30];        { Using derived string types }
  String6  = String[6];         { makes type NAPRec smaller }
  String3  = String[3];

  NAPPtr = ^NAPRec;
  NAPRec = RECORD
             Name    : String30;
             Address : String30;
             City    : String30;
             State   : String3;
             Zip     : String6;
             Next    : NAPPtr        { Points to next NAPRec }
           END;                      { in a linked list }

  NAPFile = FILE OF NAPRec;


VAR
  Ch       : Char;
  Root     : NAPPtr;
  Quit     : Boolean;



{$I YES.SRC }      { Contains Yes }


PROCEDURE ClearLines(First,Last : Integer);

VAR
  I : Integer;

BEGIN
  FOR I := First TO Last DO
    BEGIN
      GotoXY(1,I);
      ClrEOL
    END
END;



PROCEDURE ShowRecord(WorkRec : NAPRec);

VAR
  I : Integer;

BEGIN
  ClearLines(17,22);  { Clear away anything in that spot before }
  GotoXY(1,17);
  WITH WorkRec DO
    BEGIN
      Writeln('>>Name:     ',Name);
      Writeln('>>Address:  ',Address);
      Writeln('>>City:     ',City);
      Writeln('>>State:    ',State);
      Writeln('>>Zip:      ',Zip)
    END
END;

```

```

PROCEDURE CheckSpace;

VAR
  Space      : Integer;
  RealRoom   : Real;
  RecordRoom : Real;

BEGIN
  Space := MemAvail;      { MemAvail returns negative Integer for   }
                          { space larger than 32,767.  Convert }
                          { (to a real) by adding 65536 if negative }
  IF Space < 0 THEN RealRoom := 65536.0 + Space ELSE RealRoom := Space;

  RealRoom := RealRoom * 16;     { Delete this line for Z80 versions! }
                                 { MemAvail for 8086 returns 16-byte  }
                                 { paragraphs, not bytes!! }

  RecordRoom := RealRoom / SizeOf(NAPRec);
  ClearLines(2,3);
  Writeln('>>There is now room for ',RecordRoom:6:0,' records in your list.');
END;


PROCEDURE ListDispose(VAR Root : NAPPtr);

VAR
  Holder : NAPPtr;

BEGIN
  GotoXY(27,10); Write('>>Are you SURE? (Y/N): ');
  IF YES THEN
    IF Root <> Nil THEN
      REPEAT
        Holder := Root^.Next;      { First grab the next record...       }
        Dispose(Root);             { ...then dispose of the first one... }
        Root := Holder             { ...then make the next one the first }
      UNTIL Root = Nil;
  ClearLines(10,10);
  CheckSpace
END;


PROCEDURE AddRecords(VAR Root : NAPPtr);

VAR
  I       : Integer;
  Abandon : Boolean;
  WorkRec : NAPRec;
  Last    : NAPPtr;
  Current : NAPPtr;

BEGIN
  GotoXY(27,7); Write('<<Adding Records>>');
  REPEAT                  { Until user answers 'N' to "MORE?" question... }
    ClearLines(24,24);
    FillChar(WorkRec,SizeOf(WorkRec),CHR(0));  { Zero the record }
    ClearLines(9,15);
    GotoXY(1,9);
    WITH WorkRec DO          { Fill the record with good data }
      BEGIN
        Write('>>Name:     '); Readln(Name);
        Write('>>Address:  '); Readln(Address);
        Write('>>City:     '); Readln(City);
        Write('>>State:    '); Readln(State);
        Write('>>Zip:      '); Readln(Zip)
      END;
    Abandon := False;
                         { Here we traverse list to spot duplicates: }

    IF Root = Nil THEN      { If list is empty point Root to record }
      BEGIN
        New(Root);
        WorkRec.Next := Nil;  { Make sure list is terminated by Nil }
        Root^ := WorkRec;
      END
    ELSE                       { ...if there's something in list already   }
      BEGIN
        Current := Root;       { Start traverse at Root of list }
        REPEAT
          IF Current^.Name = WorkRec.Name THEN { If duplicate found }
            BEGIN
              ShowRecord(Current^);
              GotoXY(1,15);
              Write
('>>The record below duplicates the above entry''s Name.  Toss entry? (Y/N): ');
              IF Yes THEN Abandon := True ELSE Abandon := False;
              ClearLines(15,22)
            END;
          Last := Current;
          Current := Current^.Next
        UNTIL (Current = Nil) OR Abandon OR (Current^.Name > WorkRec.Name);

        IF NOT Abandon THEN               { Add WorkRec to the linked list  }
          IF Root^.Name > WorkRec.Name THEN  { New Root item!      }
            BEGIN
              New(Root);                  { Create a new dynamic NAPRec   }
              WorkRec.Next := Last;       { Point new record at old Root }
              Root^ := WorkRec            { Point new Root at WorkRec     }
            END
          ELSE
            BEGIN
              NEW(Last^.Next);            { Create a new dynamic NAPRec, }
              WorkRec.Next := Current;    { Points its Next to Current   }
              Last^.Next^ := WorkRec;     { and assign WorkRec to it     }
              CheckSpace                  { Display remaining heapspace }
            END;
      END;
    GotoXY(1,24); Write('>>Add another record to the list? (Y/N): ');
  UNTIL NOT Yes;
END;


PROCEDURE LoadList(VAR Root : NAPPtr);

VAR
```

```
  WorkName : String30;
  WorkFile : NAPFile;
  Current  : NAPPtr;
  I        : Integer;
  OK       : Boolean;

BEGIN
  Quit := False;
  REPEAT
    ClearLines(10,10);
    Write('>>Enter the Name of the file you wish to load: ');
    Readln(WorkName);
    IF Length(WorkName) = 0 THEN   { Hit (CR) only to abort LOAD }
      BEGIN
        ClearLines(10,12);
        Quit := True
      END
    ELSE
      BEGIN
        Assign(WorkFile,WorkName);
        {$I-} Reset(WorkFile); {$I+}
        IF IOResult <> 0 THEN              { 0 = OK; 255 = File Not Found }
          BEGIN
            GotoXY(1,12);
            Write('>>That file does not exist.  Please enter another.');
            OK := False
          END
        ELSE OK := True                    { OK means File Is open }
      END
    UNTIL OK OR Quit;
  IF NOT Quit THEN
    BEGIN
      ClearLines(10,12);
      Current := Root;
      IF Root = Nil THEN                   { If list is currently empty }
        BEGIN
          NEW(Root);                       { Load first record to Root^ }
          Read(WorkFile,Root^);
          Current := Root
        END                                { If list is not empty, find the end: }
      ELSE WHILE Current^.Next <> Nil DO Current := Current^.Next;
      IF Root^.Next <> Nil THEN { If file contains more than 1 record }
      REPEAT
        NEW(Current^.Next);              { Read and add records to list }
        Current := Current^.Next;        { until a record's Next field  }
        Read(WorkFile,Current^)          { comes up Nil    }
      UNTIL Current^.Next = Nil;
      CheckSpace;
      Close(WorkFile)
    END
END;


PROCEDURE ViewList(Root : NAPPtr);

VAR
  I        : Integer;
  WorkFile : NAPFile;
```

```
   Current  : NAPPtr;

BEGIN
  IF Root = Nil THEN                    { Nothing is now in the list }
    BEGIN
      GotoXY(27,18);
      Writeln('<<Your list is empty!>>');
      GotoXY(26,20);
      Write('>>Press (CR) to continue: ');
      Readln
    END
  ELSE
    BEGIN
      GotoXY(31,7); Write('<<Viewing Records>>');
      Current := Root;
      WHILE Current <> Nil DO   { Traverse and display until Nil found }
        BEGIN
          ShowRecord(Current^);
          GotoXY(1,23);
          Write('>>Press (CR) to view Next record in the list: ');
          Readln;
          Current := Current^.Next
        END;
      ClearLines(19,22)
    END
END;


PROCEDURE SaveList(Root : NAPPtr);

VAR
  WorkName : String30;
  WorkFile : NAPFile;
  Current  : NAPPtr;
  I        : Integer;

BEGIN
  GotoXY(1,10);
  Write('>>Enter the filename for saving out your list: ');
  Readln(WorkName);
  Assign(WorkFile,WorkName);   { Open the file for write access }
  Rewrite(WorkFile);
  Current := Root;
  WHILE Current <> Nil DO      { Traverse and write }
    BEGIN
      Write(WorkFile,Current^);
      Current := Current^.Next
    END;
  Close(WorkFile)
END;



BEGIN      { MAIN }
  ClrScr;
  GotoXY(28,1); Write('<<Linked List Maker>>');
  CheckSpace;
  GotoXY(17,8);  Write('---------------------------------------------');
```

```
    Root := Nil; Quit := False;
    REPEAT
      ClearLines(5,7);
      ClearLines(9,24);
      GotoXY(1,5);
      Write
      ('>>[L]oad, [A]dd record, [V]iew, [S]ave, [C]lear list, or [Q]uit: ');
      Readln(Ch);                      { Get a command }
      CASE Ch OF
       'A','a' : AddRecords(Root);  { Parse the command & perform it }
       'C','c' : ListDispose(Root);
       'L','l' : LoadList(Root);
       'S','s' : SaveList(Root);
       'V','v' : ViewList(Root);
       'Q','q' : Quit := True;
      END; { CASE }
    UNTIL Quit
  END.
```

21.4: LINKED LISTS OF DIRECTORY ENTRIES

In Chapter 20 we spent a fair amount of time exploring the uses of the DOS FIND FIRST and FIND NEXT function calls. FIND FIRST will find the first (or the only) file matching a given filespec. However, when multiple files match a given filespec in a given directory, FIND NEXT will keep popping up appropriate directory entries, one after the other. There could be three, a dozen, a hundred, or five hundred. With today's inexpensive 32MB hard disks and tomorrow's gigabyte WORM and CD ROM drives, you have to be ready to deal with a lot of files. Unless you were prepared to process the files you found on the spot (as program **Locate** from Section 20.9 does, by simply displaying their pathnames and other information to the screen), you wouldn't keep calling FIND NEXT without somehow planning to keep the directory entries it returns somewhere for later use.

Where to put them? You could define an array of **DIRRec**, but in Pascal the size of the array must be set at compile time. If you plan for a worst-case scenario of holding an array of 500 directory entries, that array will occupy close to 48K of RAM, and 64K is as large as any single Pascal data item can be. Worse, even if you only want to store three directory entries, the entire array must exist, taking up room with 497 records full of dead space.

The better solution is to string them together in a linked list on the heap, by creating a dynamically allocated **DIRRec** record on the heap for each directory entry, and connecting them with pointers. Heapspace under Turbo Pascal is very large and occupies *all* available memory beyond the end of compiler, code, and data, all the way up to 640K. This is called a "long heap," and only a few Pascal compilers support it.

The remainder of Chapter 21 will explore the uses of heap storage in managing large numbers of directory entries returned by repeated calls to FIND NEXT.

Doubly Linked Lists

Singly linked lists have a fairly serious limitation: You can traverse them only in one direction—starting from the root. To simply search a list in order to tell if a particular data item is present, traversing a list in one direction is sufficient. You might, however, wish to display the items in a linked list in ascending or descending order. You might wish to display the items in a list in a screen menu, and select one item from the list by moving a "bounce bar" of inverse video up and down the list, finally choosing one item by pressing a particular key, as done in Word Perfect Corporation's P-Edit and Word Perfect editors. For applications like these, a doubly linked list is called for.

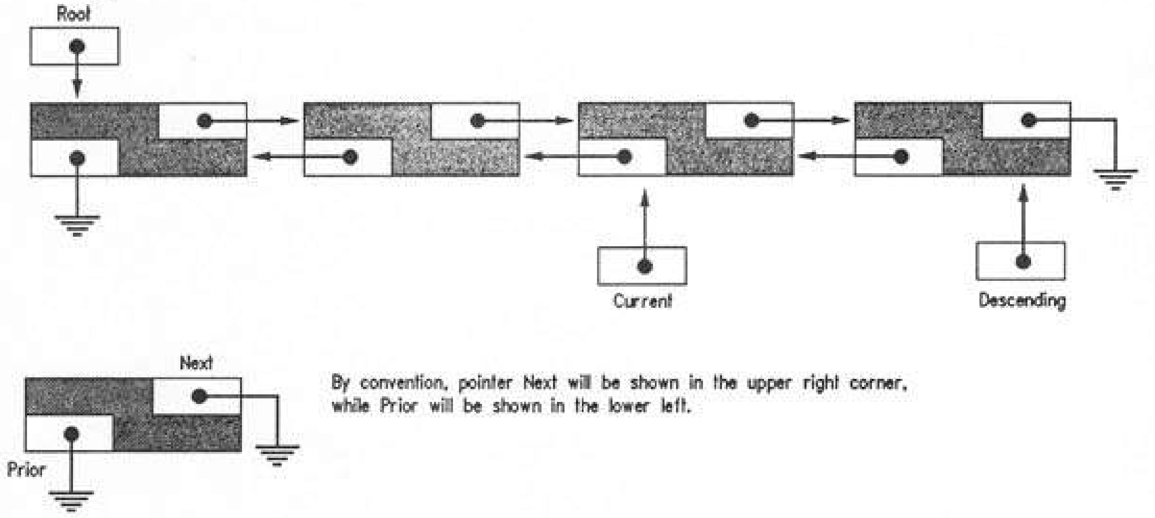
Figure 21.5 represents a doubly linked list of four items. Note that each record in the list contains two pointers instead of one, and that there is a pointer variable in the static data area pointing to each end of the list. **Root** is still there, by convention on the left, but I call the pointer at the opposite end of the list **Descending**, because to traverse a sorted list in descending order, you must begin at the end of the list *opposite* **Root**.

The two pointers within each record are called **Next** and **Prior**. The **Next** fields in the list point "away" from **Root** toward **Descending**, and the **Prior** fields point away from **Descending** toward **Root**. Assuming a sorted list with the first item pointed to by **Root** and the last by **Descending**, we now have a second way to traverse the list:

```
Current := Descending;
WHILE (Current^.Name <> 'Frodo') AND (Current <> NIL) DO
  Current := Current^.Prior;
```

Figure 21.5

A Doubly Linked List


Root
Current
Descending
Next
Prior
By convention, pointer Next will be shown in the upper right corner, while Prior will be shown in the lower left.

This **WHILE** loop begins at **Descending**'s end of the list and searches the list rootward until it either finds a record containing the name Frodo or until it reaches the record **Root^**, whose **Prior** field is set to **Nil**.

Structurally, a doubly linked list is completely symmetrical. Anything you can do in one direction you can also do in the other direction.

In the previous discussion of singly linked lists, the process of building a list was very simple because the list was not an *ordered* list. Its records were placed in the list as created without any regard for how the data in one related to the data in any of the others. Essentially, building a nonordered list is no more complex than inserting each record at the head of the list. It's rather like letting out the string of a kite.

Building an ordered list involves more overhead. For each insertion, the list must be traversed in order to find the correct place in the list for the inserted record. When the proper point is found, the new record is inserted at that point. Three possible scenarios can arise in locating the proper spot for a new record: The new record can fall at the root end of the list, at the descending end of the list, or somewhere in the middle. The code for handling each situation is different.

The code for list insertion follows. Assume that the list is to be ordered on a record field called **KeyField**. Also assume that the record to be inserted has been created through a pointer called **Holder**, and that it has already been filled with data.

```
Current := Root;
REPEAT
  IF Current^.KeyField > Holder^.KeyField THEN
    PositionFound := True ELSE PositionFound := False;
  IF NOT PositionFound THEN Current := Current^.Next
UNTIL PositionFound OR (Current = NIL);
IF PositionFound THEN { Record falls at beginning or middle }
  BEGIN
    IF Current = Root THEN      { Insert at the beginning }
      BEGIN
        Holder^.Next := Root;
        Current^.Prior := Holder;
        Root := Holder;
        Root^.Prior := NIL
      END
    ELSE      { Insert in the middle, right before Current^ }
      BEGIN
        Holder^.Next := Current;
        Holder^.Prior := Current^.Prior;
        Current^.Prior^.Next := Holder;
        Current^.Prior := Holder
      END
  END
ELSE                  { Record falls at the end }
  BEGIN
    Descending^.Next := Holder;
  Descending^.Next^.Prior := Descending;
  Descending := Descending^.Next;
  Descending^.Next := NIL
END;
```

The **REPEAT** statement accomplishes the traversal of the linked list. At each record in the list, the data in **Holder^** is compared to the data in **Current^**. If **Current^** is not found to be greater than **Holder^**, then the pointer **Current** is bumped to point to the next record in the list. Boolean flag **PositionFound** simply marks when the code decides that it knows where the record should go.

This traversal could apply to a singly linked list as well as a doubly linked list, since it is only testing in one direction. The **Prior** field of the records in the list has no function during traversal.

Inserting records into the list at any point must be done with some care. The order in which the pointers are modified is important. Changing the order could "break" the list, and, while doubly linked lists are easier to mend than singly linked lists (because there are two independent threads holding the list together), there's no sense in doing more than you have to in managing the list.

[Inserting a record at the beginning of the list first involves pointing **Holder^** at **Root^**; this places the inserted record at the head of the list. The **Prior** pointer in the record that used to be the first record in the list must be pointed "back" at the new head of the list (its previous value was **Nil**). **Root** must now be moved so that it points to the new head of the list. This is done by assigning it **Holder**'s value. Finally, the **Prior** field of the new head record is set to **Nil**, as it now points to nothing, there being no record prior to it.]

Appending a record to the opposite end of the list is the mirror image of that process. The tail end record's **Next** pointer is pointed to the new record. The **Prior** pointer of the new end record is pointed back at the old end record. **Descending** is moved to point to the new end of the list. Finally, the **Next** field of the new end record is set to **Nil**, since there is no next record after the end record.

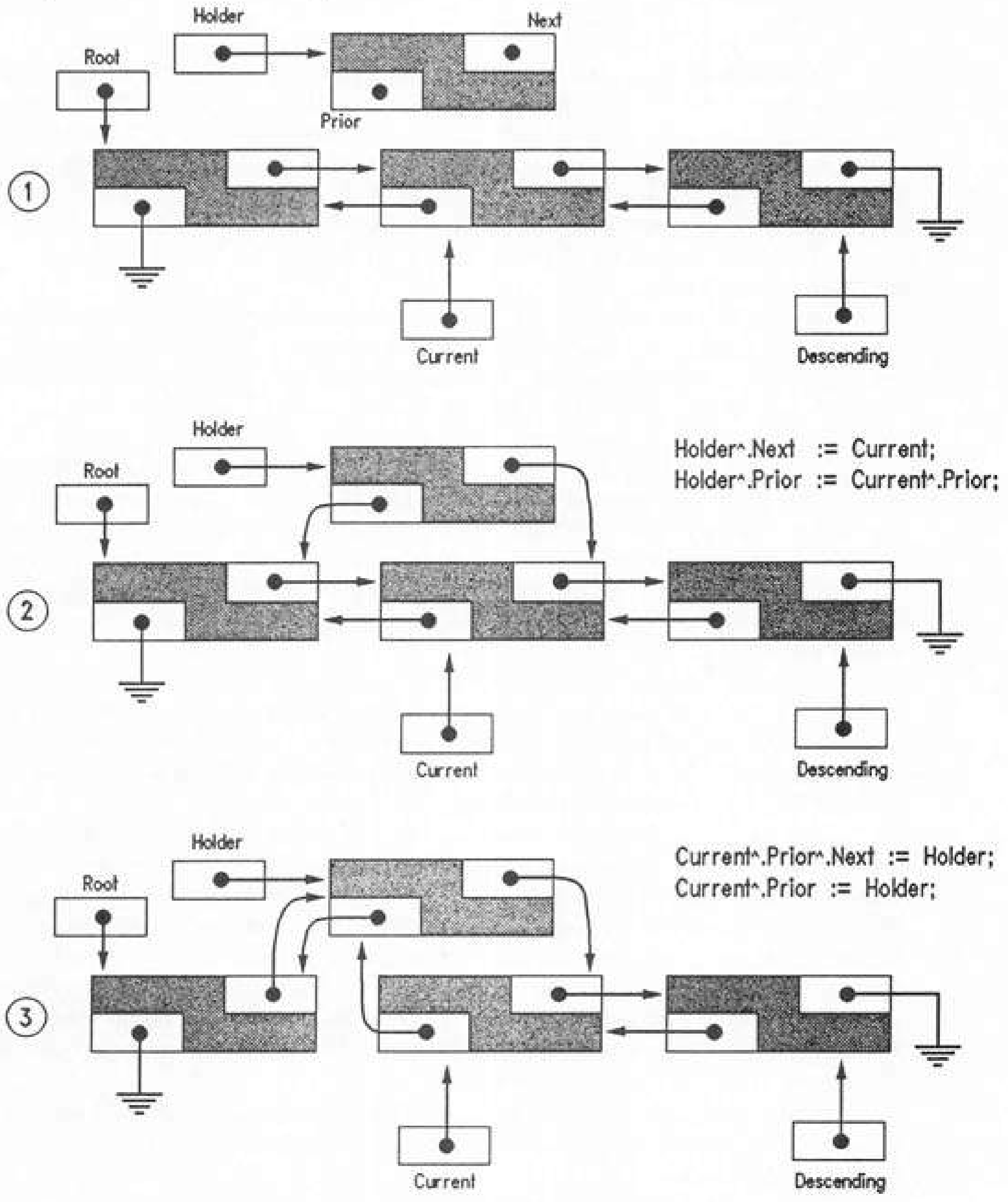
Inserting a record into the middle of a doubly linked list is summarized in Figure 21.6. Bullet 1 shows the situation before the insertion: A new record pointed to by **Holder** is ready for insertion. The next step, shown by bullet 2, involves pointing the **Prior** and **Next** pointers in the new record to their appropriate new target records in the list. At this point it might seem that the new record has somehow become attached to the list, but not so. If something happened to **Holder**, the new record would be completely inaccessible, because, although it points to two records in the list, nothing points to *it*. Pointers are strictly a one-way street. The target of a pointer cannot look "back along the pointer" to identify who is pointing to it.

Bullet 3 completes the process, by repointing the two records between which the new record falls back to the new record. All in all, four records must be altered to insert the new record.

The process of deleting a record in the middle of a list is considerably simpler, as shown in Figure 21.7. Two pointers are required, because deleting an entire record

Figure 21.6

Inserting a Record into a Doubly Linked List



```
Holder^.Next  := Current;
Holder^.Prior := Current^.Prior;
```

```
Current^.Prior^.Next := Holder;
Current^.Prior := Holder;
```

breaks *both* threads running through the list. The record is deleted by way of the **Dispose** procedure, and it can be disposed of from either the **Current** side or the **Holder** side:

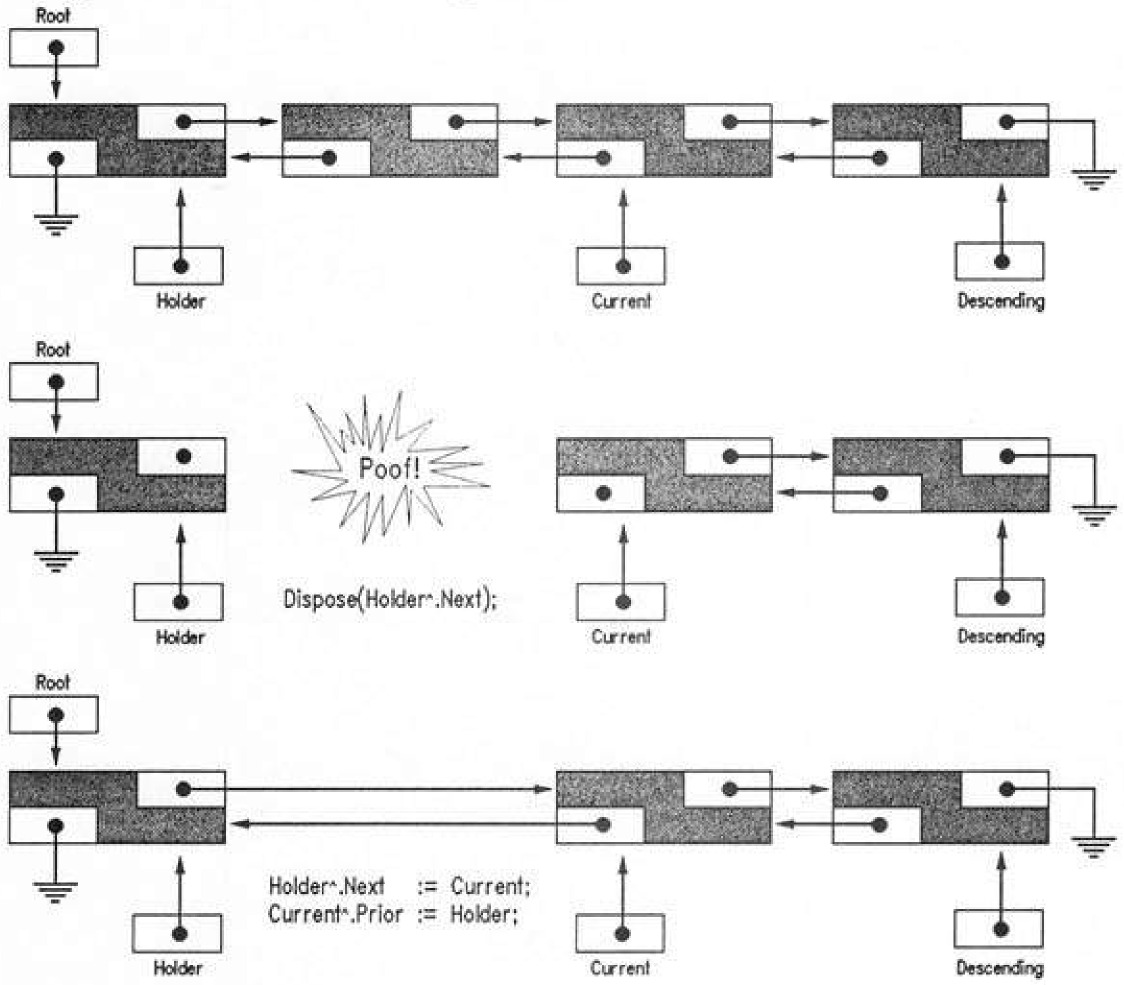
```
Dispose(Current^.Prior);
Dispose(Holder^.Next);
```

After that, "healing" the break simply involves pointing the two records alongside the break to one another:

```
Holder^.Next := Current;
Current^.Prior := Holder;
```

Figure 21.7

Deleting a Record from within a Doubly Linked List



Creating a List of Directory Entries

Because of Pascal's strong typing restrictions, it is impossible to create a "black box" linked list handler that creates and manages linked lists of arbitrarily typed records. That being the case, you will have to build linked-list management into any application that requires it. Understanding the concept thoroughly is crucial to implementing it. And the best way to understand it is to develop a useful toolkit routine that uses it.

Procedure **GetDirectory** returns two pointers to a doubly linked list of directory entries matching a given filespec. The two pointers are called **Ascending** and **Descending**, but in fact **Ascending** is our old friend **Root** with a new name. Figure 21.5 in particular will be helpful in understanding the operating of **GetDirectory**.

```
{->>>>GetDirectory<<<<-------------------------------------------}
{                                                                 }
{ Filename: GETDIR.SRC -- Last modified 7/2/88                    }
{                                                                 }
{ This routine returns a pointer to a linked list of type         }
{ DIRRec, which must have been previously defined this way,       }
{ along with pointer type DIRPtr to point to it:                  }
{                                                                 }
{ DIRPtr = ^DIRRec;                                               }
{ DIRRec = RECORD                                                 }
{             FileName  : String15;                               }
{             Attrib    : Byte;                                   }
{             FileSize  : LongInt;                                }
{             TimeStamp : TimeRec;                                }
{             DateStamp : DateRec;                                }
{             Prior     : DIRPtr;                                 }
{             Next      : DIRPtr;                                 }
{           END;                                                  }
{                                                                 }
{ The linked list will contain a record for every file in the     }
{ current directory.  Since the linked list is out in heap,       }
{ your directory data takes up NO space in your data segment.     }
{ If there are no files in the current directory, the pointer     }
{ returned is equal to NIL.                                       }
{                                                                 }
{ The types TimeRec and DateRec must be defined prior to using    }
{ GetDirectory.  String80 & DTAToDIR must be defined as well.     }
{                                                                 }
{                                                                 }
{                                                                 }
{-----------------------------------------------------------------}

PROCEDURE GetDirectory(Filespec    : String80;
                       Sorted      : Boolean;
                       SortOnName  : Boolean;
                       VAR Ascending  : DIRPtr;
                       VAR Descending : DIRPtr);

TYPE
  String9 = String[9];

VAR
```

```
  I          : Integer;
  FindError  : Integer;
  Regs       : Registers;
  OurDTA     : SearchRec;
  Root       : DIRPtr;
  Current    : DIRPtr;
  Last       : DIRPtr;
  Holder     : DIRPtr;
  PositionFound : Boolean;


FUNCTION LaterThan(LeftEntry,RightEntry : DirPtr) : Boolean;

BEGIN
  IF LeftEntry^.DateStamp.DateComp > RightEntry^.DateStamp.DateComp THEN
    LaterThan := True
  ELSE
    IF (LeftEntry^.DateStamp.DateComp = RightEntry^.DateStamp.DateComp)
        AND
       (LeftEntry^.TimeStamp.TimeComp > RightEntry^.TimeStamp.TimeComp)
    THEN LaterThan := True
    ELSE LaterThan := False
END;


PROCEDURE AppendToEnd(VAR Holder,Descending : DIRPtr);

BEGIN
  Descending^.Next := Holder;     { Add record to end of list }
  Descending^.Next^.Prior := Descending;  { Set reverse pointer }
  Descending := Descending^.Next;    { Bump Current to next record }
END;



BEGIN  { GetDir }
  FindFirst(FileSpec,$16,OurDTA);        { Make FIND FIRST DOS call... }
  FindError := DOSError;
  IF FindError = 2 THEN  { No files found to match FileSpec }
    BEGIN
      Ascending := NIL;      { Both linked list pointers are NIL }
      Descending := NIL
    END
  ELSE                    { There was at least one file found, so... }
    BEGIN
      New(Root);             { Create a record for the first find }
      DTAtoDIR(Root^);       { Convert first find to DIR format }
      Current := Root;       { The current record is now the root record }
      Descending := Root;    { And also the last record in the list! }
      IF FindError <> 18 THEN
        REPEAT
          FindNext(OurDTA);        { Make FIND NEXT DOS call }
          FindError := DOSError;
          IF FindError <> 18 THEN { More entries exist }
            BEGIN
              New(Holder);         { Create a record with temporary pointer }
              DTAtoDIR(Holder^);   { Convert additional finds to DIR format }
              { Sorted and unsorted lists are constructed differently. }
              { If we're building a sorted list we have to scan it for }
              { each entry to find the proper place in the list.  For  }
              { unsorted lists we just hang the latest found entry on  }
              { the end of the list and bump Current to the Next. }
              IF Sorted THEN
                BEGIN
                  Current := Root;    { Traverse list to find insert spot: }
                  REPEAT
                    IF SortOnName THEN     { To sort list on file name }
                      IF Current^.FileName > Holder^.FileName THEN
                        PositionFound := True ELSE PositionFound := False
                    ELSE                   { To sort list by time/date }
                      IF LaterThan(Current,Holder) THEN
                        PositionFound := True ELSE PositionFound := False;
                    IF NOT PositionFound THEN
                      Current := Current^.Next;  { Bump to next item }
                  UNTIL (Current = NIL) OR PositionFound;
                  { When PositionFound becomes True, the record needs }
                  { to be inserted in the list BEFORE Current^-- }
                  { This needs to be done differently if Current^ is }
                  { at the head of the list. (i.e., Current = Root)}
                  IF PositionFound THEN  { Insert at beginning... }
                    BEGIN                { ...or in the middle somewhere }
                      { NOTE:  DO NOT change the order of the }
                      { pointer assignments in the following }
                      { IF/THEN/ELSE statement!! }
                      IF Current = Root THEN  { Insert at beginning }
                        BEGIN
                          Holder^.Next := Root;
                          Current^.Prior := Holder;
                          Root := Holder;
                        END
                      ELSE    { Insert in the middle: }
                        BEGIN
                          Holder^.Next  := Current;
                          Holder^.Prior := Current^.Prior;
                          Current^.Prior^.Next := Holder;
                          Current^.Prior := Holder
                        END
                    END
                  ELSE  { The new record belongs at the end of the list }
                    AppendToEnd(Holder,Descending)
                END
              ELSE   { If no sort, we add the record to the end of the list: }
                AppendToEnd(Holder,Descending)
            END
        UNTIL FindError = 18;
      Ascending := Root
    END
END;  {GetDirectory}
```

The full search spec, including the path and disk specifier, is passed to **GetDirectory** in parameter **FileSpec**. As with any use of DOS's FIND FIRST/FIND NEXT sequence, the search spec must be assembled into an ASCIIZ string, and the string's address loaded into registers DS and DX. The attribute byte is also set, but by choice we're intending to read every kind of file except for the volume label: hidden files, system files, subdirectories, and of course ordinary DOS data files. Taken together, the hidden file bit, the system file bit, and the subdirectory bit represent a binary value of $16, which is passed to DOS in register CL.

If DOS error messages 2 or 18 are returned by FIND FIRST, the directory is empty, and both **Ascending** and **Descending** are returned with a value of **Nil**. Otherwise, a new **DIRRec** record is created as the root of the new list, and filled with information returned by DOS in the DTA. Repeated calls by FIND NEXT will read the remainder of the directory, until an error message indicates no further entries to be read. As each directory entry is read, **GetDirectory** adds the entry to the doubly linked list it is building.

GetDirectory offers two different ways to build the list: In the order the directory entries physically appear on the disk, or in sorted order. The caller's preference is passed to **GetDirectory** in Boolean parameter **Sorted**. If sorted order is selected (**Sorted** set to **True**), there is the further choice of sorting the list on either the name of the file or subdirectory (**SortOnName = True**), or on the time and date of last modification (**SortOnName = False**).

Determining which of two string records is greater than another in alphanumeric sort order is simple: They can be compared via Pascal's Boolean operators. Determining which of two time/date combinations is the earlier is a little trickier, even with the time and date present as single integer quantities in variables **TimeComp** and **DateComp** (see Section 20.7). If the "left" date is greater than the "right" date, then the time does not have to be taken into account, but if the dates are equal, the time stamps must also be compared. To aid the readability of the procedure, this process was set off in a short function called **LaterThan**, which accepts pointers to **DIRRec**s and returns a Boolean value of **True** if the left entry is greater than the right entry as passed in the parameter line.

The logic of adding a record to the list is very close to the code example given on page 446, with some additional conditional logic to allow ordering the list on either the name or the time/date fields. Read the code example, and compare it to the actual code for **GetDirectory** starting at line 105.

A File Size Tally Utility

As a practical example of **GetDirectory** in use, let me present **Spacer**, a utility that does something I have always wished DOS DIR would do: Present a tally of the sizes of the displayed files. Many times I've had to clear out a couple of megabytes from a 20- megabyte hard disk (which always seems to stay stuck at 18.5 megabytes full, no matter what I do!) and I have had to become a good eyeball estimator of the space I'll save if I just do an ERASE *.BAK or something along those lines. **Spacer** eliminates the guesswork. To find out the space taken up in the current directory by backup files (or any other kind of files you specify), type:

```
C  > SPACER *.BAK

LOCATE   .BAK      11503  01/14/87  11:15a
DIRSTRIN.BAK        2623  01/13/87   7:53p
SPACER   .BAK       6055  01/14/87   9:33a
GETDIR   .BAK       8595  01/14/87   9:31a
DTEST    .BAK       5328  01/15/87   5:54p
DISPDIR .BAK         443  01/15/87   5:59p
DTATODIR.BAK        2701  01/13/87  11:18a

Total space occupied by these files is     37248 bytes.
```

Spacer works by generating a doubly linked list of the requested files, using **GetDirectory**. It then displays the list, and keeps a running total of the sizes of the files as it displays them. When the display has been completed, **Spacer** adds an additional line giving the total number of bytes of disk space occupied by the files whose names have just been displayed.

At first glance, you might wonder why a linked list needs to be built at all. After all, if program **Locate** from the previous section displays files from directories, why couldn't you just add a little code to keep a running tally of files displayed from **Locate**? Well, of course, you could, and it's probably a worthwhile improvement on **Locate**. But when you display them as you find them, you lose the ability to display them sorted by name or by last-modified time and date. You can only display them in the order in which the directory entries physically appear on the disk. For that you need a linked list, and to offer the choice of displaying them either in ascending or descending order by your chosen sort field, you need a *doubly* linked list.

```
{--------------------------------------------------------------}
{                            SPACER                            }
{                                                              }
{            Directory lister with file size tally             }
{                                                              }
{                             by Jeff Duntemann                }
{                             Turbo Pascal 5.00                }
{                             Last update 7/2/88               }
{                                                              }
{ This utility functions similarly to DOS DIR in that it       }
{ displays a directory of files in a subdirectory, but unlike }
{ DIR it keeps a running total of the size of the files        }
{ displayed.  It exists mostly to demonstrate the generation   }
{ of a linked list of directory entries through the procedure }
{ GetDirectory.  Ideally, it should be expanded into a         }
{ utility similar to SWEEP.                                    }
{                                                              }
{                                                              }
{                                                              }
{--------------------------------------------------------------}

PROGRAM Spacer;

USES DOS;
```

```

CONST
  SortByName = True;
  SortByDate = False;

TYPE
 String80 = String[80];
 String15 = String[15];
 DTAPtr   = ^SearchRec;

{$I TIMEREC.DEF}     { Described in Section 20.6 }
{$I DATEREC.DEF}     { Described in Section 20.6 }
{$I DIRREC.DEF}      { Described in Section 20.7 }


VAR
  Parms      : Byte;
  SpaceTaken : Real;
  RunUp      : DIRPtr;
  RunDown    : DIRPtr;
  Current    : DIRPtr;
  FileSpec   : String80;
  WorkString : String80;
  Sorted     : Boolean;
  SortSpec   : Boolean;
  Ascending  : Boolean;
  I          : Integer;


{$I DAYOWEEK.SRC}    { Described in Section 20.6 }
{$I CALCDATE.SRC}    { Described in Section 20.6 }
{$I CALCTIME.SRC}    { Described in Section 20.6 }
{$I DTATODIR.SRC}    { Described in Section 20.7 }
{$I GETDIR.SRC}      { Described in Section 21.4 }
{$I DIRSTRIN.SRC}    { Described in Section 20.7 }



BEGIN
  Sorted := False;            { Set default values }
  SortSpec := SortByName;
  Ascending := True;
  Parms := ORD(ParamCount);  { Convert parm count to ordinal value }
  CASE Parms OF
    0 :
    BEGIN
      Writeln('>>SPACER<<  V2.00  By Jeff Duntemann');
      Writeln('             From the book, COMPLETE TURBO PASCAL 5.0');
      Writeln('             Scott, Foresman & Co. 1988');
      Writeln('             ISBN 0-673-38355-5');
      Writeln;
      Writeln('This program displays ALL files matching a given filespec.');
      Writeln('Hidden and system files are not immune.');
      Writeln('Additionally, it will add up the cumulative file sizes of');
      Writeln('the files matching the filespec, so you can tell how much');
      Writeln('space files in a given subdirectory subtend, or how much');
      Writeln('space you have invested in .PAS files, and so on.');
      Writeln;
  Writeln('CALLING SYNTAX:');
  Writeln;
  Writeln('SPACER <filespec> N|D A|D');
  Writeln;
  Writeln('where <filespec> is a legal DOS filespec, including wildcards.');
  Writeln('The second parameter is either N or D:');
  Writeln('N indicates sort by file name;');
  Writeln('D indicates sort by time and date stamp.');
  Writeln('If not given, entries are displayed in physical order.');
  Writeln;
  Writeln('The third parameter is either A or D:');
  Writeln('A indicates ascending order of display;');
  Writeln('D indicates descending order of display.');
  Writeln('If not given, sort is ascending.');
  Writeln;
  Writeln('For example:');
  Writeln;
  Writeln('SPACER *.PAS N');
  Writeln('  will display all files with the .PAS extension,');
  Writeln('  in ascending sorted order by file name.  Or,');
  Writeln;
  Writeln('SPACER *.PAS D D');
  Writeln('  will display all files with the .PAS extension,');
  Writeln('  in descending order by last-modification date.');
  Halt;
END;
1 : FileSpec := ParamStr(1);
2 : BEGIN
      Sorted := True;
      FileSpec := ParamStr(1);
      WorkString := ParamStr(2);
      CASE UpCase(WorkString[1]) OF
        'D' : SortSpec := SortByDate;
        'N' : SortSpec := SortByName;
        ELSE Sorted := False
      END
    END;
3 : BEGIN
      Sorted := True;
      FileSpec := ParamStr(1);
      WorkString := ParamStr(2);
      CASE UpCase(WorkString[1]) OF
        'D' : SortSpec := SortByDate;
        'N' : SortSpec := SortByName;
        ELSE Sorted := False
      END;
      IF Sorted THEN
        BEGIN
          WorkString := ParamStr(3);
          CASE UpCase(WorkString[1]) OF
            'A' : Ascending := True;
            'D' : Ascending := False;
            ELSE Ascending := True
          END
        END
    END;
END; { CASE }
{ Now we actually go out and build a linked list of directory entries, }
```

```
    { based on the parms we have parsed out of the command line: }
    GetDirectory(FileSpec,Sorted,SortSpec,RunUp,RunDown);
    IF Ascending THEN Current := RunUp
      ELSE Current := RunDown;
    IF Current = Nil THEN Writeln('No files found.')
      ELSE
        BEGIN
          SpaceTaken := 0.0;
          IF Ascending THEN
            WHILE Current <> NIL DO
              BEGIN
                Writeln(DirToString(Current^));
                SpaceTaken := SpaceTaken + Current^.FileSize;
                Current := Current^.Next
              END
          ELSE
            WHILE Current <> NIL DO
              BEGIN
                Writeln(DirToString(Current^));
                SpaceTaken := SpaceTaken + Current^.FileSize;
                Current := Current^.Prior
              END;
          Writeln;
          Writeln
          ('Total space occupied by these files is ',SpaceTaken:9:0,' bytes.');
        END
END.
```

One trick to keep in mind in using **Spacer** is that it sends its output to DOS standard output device, allowing output redirection. You can take **Spacer**'s files list and send it to a disk file instead of to the screen:

```
C:\>SPACER *.PAS > PASFILES.DIR
```

This command creates a text file named **PASFILES.DIR** and writes the exact same text output to the file that you would ordinarily see on the screen.

Disposing of a List of Directory Entries

The lists of directory entries created by **GetDirectory** exist on the heap, and take up heap space. For utilities like **Spacer**, this is not especially important, because the utility creates the linked list, displays it, and then returns to DOS. For more complicated applications that need to reuse heap space, some method of disposing of the directory lists is required.

Such a routine, by this time, should seem almost trivial. Singly and doubly linked lists can be disposed of in nearly identical fashion, since only one thread is required to tether the list while the rootmost entry is disposed of. Procedure **DisposeOfDirectory** will do the job quickly and safely.

```
{->>>>DisposeOfDirectory<<<<------------------------------------}
{                                                                }
{ Filename : DISPDIR.SRC -- Last Modified 7/11/88                }
{                                                                }
{ This routine disposes of lists of DIRRec records as built by  }
{ GetDirectory.  Type DIRRec and DIRPtr must be defined prior   }
{ to its inclusion.                                              }
{                                                                }
{                                                                }
{                                                                }
{----------------------------------------------------------------}

PROCEDURE DisposeOfDirectory(RootPointer : DIRPtr);

VAR
  Holder : DIRPtr;

BEGIN
  IF RootPointer <> NIL THEN      { Can't dispose if no list! }
  REPEAT
    Holder := RootPointer^.Next;  { Grab the next record. }
    Dispose(RootPointer);         { Dispose of the first... }
    RootPointer := Holder         { ...and make the next the first... }
  UNTIL RootPointer = NIL         { ...until the list is all gone. }
END;
```

21.5: THE HEAP ERROR FUNCTION

Ordinarily, if you request allocation of a block of memory on the heap through a call to **New** or **GetMem** and there is not enough memory in any single block to satisfy the request, a fatal runtime error occurs and your program aborts to DOS or the Turbo Pascal environment. This can always be avoided by using **MaxAvail** before each attempted allocation. Turbo Pascal allows slightly more sophisticated handling of such an error by installing a custom heap error function that takes control in the event of a heap overflow.

Why bother? In a sense, it allows you to give **New** or **GetMem** a "second chance" in case they fail at allocating a memory block. Once you have installed a heap error handler, the handler will get control whenever a heap memory allocation attempt fails. The heap error handler is passed the size of the block that was requested. You then can have the handler attempt to free up enough memory on the heap to satisfy the request. If the heap error handler is successful, **New** or **GetMem** can then automatically be called again to retry the heap allocation.

How the heap memory is to be freed up is your decision, and will probably be application-dependent. You might devise a "garbage collector" routine that packs all existing dynamic variables onto the heap, squeezing out wasted "holes" in heap memory. This is difficult stuff, since any brute force method will be very slow.

The easiest route would be to ask the user what data might simply be thrown away, and then disposing of that data. If you are designing an application that makes heavy use of the heap, you might keep this possibility in mind.

The heap error handler must be declared this way:

```
{$F+} FUNCTION <name> (Size : Word) : Integer; {$F-}
```

The actual name of the function is up to you; it could be **HeapErrorHandler**, **TrashMan**, or whatever you like. But it must have one **Word** parameter, it must return a value of type **Integer**, and it must be bracketed by **$F+** and **$F−** commands so that it will be invocable via an 8086 FAR call.

You install the heap error handler by assigning its address to a predefined generic pointer variable called **HeapErr**:

```
HeapErr := @TrashMan;
```

When a heap overflow error occurs, Turbo Pascal's runtime code transfers control to the address stored in **HeapErr**, assuming a **Word** parameter on the stack. This parameter will contain the size in bytes of the memory block that was requested during the failed allocation attempt. The heap error handler's mission, (should it decide to accept it) is to free up that much space on the heap.

It should try, and one would hope that it would succeed. There's no guarantee of that, of course, so provision is made to pass a value *back* to the Turbo Pascal runtime code that indicates whether the second chance was successful or not. The value is in fact the function return value, and before terminating the heap error handler should assign one of three values to its return value:

0 *Failed;* issue a runtime error and halt. This is the default, and will occur on every heap error unless you install a replacement heap error handler.

1 *Failed;* assign **Nil** to the pointer passed with the failed allocation attempt. The **Nil** pointer will indicate that the allocation attempt failed, and no runtime error will be issued.

2 *Successful;* the amount of memory passed to the error handler in the parameter **Size** was freed by the error handler. This return value signals to the Turbo Pascal runtime code that the **New** or **GetMem** heap allocation should be attempted again.

22 The Borland Graphics Interface (BGI)

There are as many ways to display graphics images as there are to constructing tribal lays (as Mr. Kipling might have said) and each and every one of them is important to someone, somewhere. It would be useful and lovely to support every one of those diverse graphics boards from a high-level language, but up until now the problem of writing fast drivers for many different boards has kept Turbo Pascal graphics limited to IBM's Color Graphics Adapter and its clones.

Those days are past. With Turbo Pascal 4.0, Borland has released a device-independent graphics system called the Borland Graphics Interface (BGI). In its first release, the BGI supported all IBM graphics adapters including the CGA, EGA, VGA, MCGA, and PC3270 graphics adapters. The initial release also supported the popular Hercules Graphics Adapter and the ATT 400-line graphics board found in ATT machines. Version 5.0 added support for IBM's 8514 graphics adapter and 256 color VGA mode.

22.1: UNDERSTANDING GRAPHICS, BORLAND-STYLE

The schism between text and graphics displays is now well understood: In a text display, a coarse grid (typically 25 × 80 but possibly larger) contains patterns of *characters*. These include the familiar letters, digits, and punctuation marks, but may also include the many special characters supported by the IBM PC: foreign language and mathematical characters, box-drawing characters, halftones and bargraph elements. The key here is that characters are indivisible atoms from which screen displays are built. The characters themselves appear on the screen as patterns of tiny dots, but the individual dots *within* a character may not be changed or otherwise controlled.

A graphics display, by contrast, is a much finer grid of individual illuminated dots called *pixels*. Here, the pixels are the indivisible atoms from which other figures are constructed. These other figures may be anything at all, but in a typical graphics software system there are procedures for drawing lines, rectangles, circles, arcs, and polygons, and more complex figures are constructed from those simpler elements. Text characters may be built from pixels, but with the greater control over individual dots, a broader selection of character styles is possible. Collections of stylistically similar character figures are called *fonts*. Furthermore, because of the complete pixel-by-pixel control over character size and position, a graphics font may be "proportionally spaced;" that is, characters take up only the horizontal space they need. An i is narrower than a w, and so on.

Any piece of hardware that controls the display of pixels on a CRT screen or display panel of some kind is called a graphics *device*. The number of the pixels in the grid determines a graphics device's *resolution*. The popular Hercules Graphics Adapter, for example, displays a grid of pixels 720 wide by 348 high. We say that its resolution is 720 X 348. The Hercules device displays only two *colors*. On a green screen these might be called "green and black," since a pixel on a Hercules screen is either illuminated or not illuminated. On the other hand, some people run the Hercules device with an amber or white screen, so it makes more sense to call the Hercules a *monochrome* device, and think of its two colors as "on" and "off."

On devices that work with color monitors, pixels may be one of several colors in a group called a *palette*. The IBM Color Graphics Adapter supports two different palettes: One contains the colors red, green, and yellow; the other contains the colors cyan, magenta, and white. The IBM Enhanced Graphics Adapter is a more complex device that supports a palette of 16 different colors.

A single graphics device may be capable of displaying different resolutions and different color palettes, depending on how it is initialized by a program. Each different resolution and color palette combination is called a graphics *mode*. IBM's original Color Graphics Adapter, for example, has two modes: One has a resolution of 320 × 200 and a palette of four colors, while the other has a resolution of 640 × 200 and a palette of only two colors.

Most display devices these days are capable of operating in both text mode and graphics mode, by turns. The default is nearly always text mode, with graphics mode looked upon as something a little more special. This is why we say we get "into" and "out of" graphics mode; text mode being the mode in which most work is generally done. The Turbo Pascal compiler and environment work only in text mode, but may produce .EXE program files that work in text mode, in graphics mode, or switch from one to the other as their work requires.

The Limits of Device Independence

Supporting more than one graphics device has been possible all along with the help of some third-party graphics software libraries or Borland's own Turbo Pascal Grafix Toolbox. The special trick performed by the BGI is that it detects *at runtime* what graphics device is installed in the host machine and loads an appropriate driver at that time. In other words, the very same .EXE file you create can run (with a little care) identical graphics on many different graphics devices *without recompiling*. This is why the BGI is called a "device-independent" graphics system.

The device drivers are separate DOS files with a .BGI extension. Your program must know where the drivers will be stored at compile time, but it need not know directly which driver will be used. The BGI is not *quite* device-independent in that the resolution of the individual graphics devices must be taken into account by the programmer. In other words, some graphics devices (like IBM's CGA) have only a 200-pixel vertical resolution, while devices like the EGA have a 350-pixel vertical resolution. Your program can query the BGI to determine what resolution the currently loaded driver can handle, and it must adjust its drawing calculations accordingly. For example, if you want to draw a line around the entire screen, your program must be prepared to draw vertical lines for the sides 200 pixels high for the CGA and 350 pixels high for the EGA. There are completely device-independent graphics systems that write all graphics to a virtual screen using "world coordinates" that are scaled to fit whatever resolution the designated output device can offer. The BGI does not go quite that far in its current release.

22.2: GETTING INTO AND OUT OF GRAPHICS MODE

Getting into graphics mode requires knowing what graphics device is installed in the computer executing a Turbo Pascal program. You have the option of explicitly specifying which graphics device, or letting the BGI determine which device is installed. There is also the issue of a graphics mode: If you specify a specific graphics device, you must also specify a graphics mode that is valid for the installed graphics device. If you let the BGI determine what device is installed in the system, the BGI will also select a mode. Your software can query this mode, and change to another mode if the mode selected by the BGI is not the one you wish to use for a given graphics device.

Letting the BGI Detect the Graphics Display Device

The easiest way to get into graphics mode is to let the BGI go out and take a look at the hardware to see what sort of display board you have installed, and then automatically initialize the highest-resolution mode valid for that board. Directing the BGI to detect the installed graphics device is done with the **DetectGraph** procedure:

```
PROCEDURE DetectGraph(VAR GraphDriver : Integer;
                      VAR GraphMode   : Integer);
```

To work correctly, **DetectGraph** must be passed the value 0 in the **GraphDriver** parameter. The best way to do this is to use the constant **Detect**, which is defined in the **Graph** unit with a value of 0:

```
GraphDriver := Detect;
DetectGraph(GraphDriver,GraphMode);
```

Keep in mind that while constant **Detect** is predefined, the variables **GraphDriver** and **GraphMode** are *not*, and they or some other two integer variables must be defined in your program for passing to **DetectGraph**.

DetectGraph returns the driver code for the installed display board in **GraphDriver** and the highest-resolution mode valid for that board in **GraphMode**. These codes are given in Tables 22.1 and 22.2.

The procedure that actually initializes the BGI is **InitGraph**:

```
PROCEDURE InitGraph(VAR GraphDriver : Integer;
                    VAR GraphMode   : Integer;
                    DriverPath      : String);
```

The variable passed to **InitGraph** as the **GraphDriver** parameter must contain a value from 0 through 10. The **Graph** unit contains a set of named constants for these values to help you in remembering which is which and what each means (Table 22.1).

Table 22.1
BGI Driver Codes for Device Initialization

Constant	*Value*	*Meaning*
Detect	= 0	Tells the BGI to attempt device detection
CGA	= 1	IBM Color Graphics Adapter (or clone)
MCGA	= 2	IBM Multi Color Graphics Array
EGA	= 3	IBM EGA (or clone) with 256K RAM
EGA64	= 4	IBM EGA with 64K RAM
EGAMono	= 5	IBM 256K EGA connected to monochrome screen
IBM8514	= 6	IBM 8514 graphics card (Version 5.0 only)
HercMono	= 7	Hercules Graphics Adapter
ATT400	= 8	AT&T 400-line display adapter
VGA	= 9	IBM Virtual Graphics Array (or clone)
PC3270	= 10	IBM 3270 PC display adapter

Other graphics devices may have been added to the driver collection by the time you read this book; this, after all, is what device-independent graphics was designed to make possible. Check the README file on your Turbo Pascal distribution disk to see if there have been any late additions.

The variable passed to **InitGraph** as the **GraphMode** parameter specifies the graphics mode to use, and must contain a value representing a valid BGI graphics mode for which a driver is available. In most cases, you will simply carry over a value from a previous call to **DetectGraph**, which detected the installed graphics device. If you specify a specific driver in **GraphDriver** and a specific graphics mode in **GraphMode**, the mode specified in **GraphMode** must be legal for the driver specified in **GraphDriver**. As with the driver specifier values, the **Graph** unit contains predefined constants for each legal mode for each supported driver. The constants and their values are shown in Table 22.2.

The modes **EGA64Lo** and **EGA64Hi** operate with early IBM-built EGA boards that had only 64K of graphics memory installed. Note that the IBM 8514 driver is not present in Turbo Pascal 4.0. Also, the IBM8514 cannot be autodetected by the BGI; it will be detected as a VGA. To use the 8514 driver, you must explicitly specify the 8514 driver by passing the constant **IBM8514** to **GraphDriver**.

The **DriverPath** parameter contains either a null string, indicating that the BGI should expect to find its device drivers in the current directory; or else the full pathname of the directory where the drivers are stored.

Something important to keep in mind: **InitGraph**'s parameters **GraphDriver** and **GraphMode** are **VAR** parameters. The **Graph** unit contains all these handy predefined constants summarized in the last two tables, but *you can't pass the constants in* ***GraphDriver*** *or* ***GraphMode***. **VAR** parameters cannot be constants. You must first assign one of the constants to an integer variable and pass the variable in **GraphDriver** or **GraphMode** instead:

```
VAR
  CurrentDriver, CurrentMode : Integer;

CurrentDriver := MCGA;
CurrentMode   := MCGAC2;

InitGraph(CurrentDriver,CurrentMode,'C:\TURBO\BGI');
```

BGI Initialization Errors

The BGI will do its best to initialize itself and the graphics system when you invoke **InitGraph**. Your programs need to be able to deal with the possibility that something will go wrong. Possible problems include these:

- You try to detect a graphics device and no graphics device is installed in the system.
- The BGI cannot find its drivers.

Table 22.2
BGI Mode Codes for Device Initialization

Constant	*Value*		*Meaning*	
CGAC1	= 0	320 × 200;	Red/Yellow/Green	1 page
CGAC2	= 1	320 × 200;	Cyan/Magenta/White	1 page
CGAHi	= 2	640 × 200;	2 color	1 page
MCGAC1	= 0	320 × 200;	Red/Yellow/Green	1 page
MCGAC2	= 1	320 × 200;	Cyan/Magenta/White	1 page
MCGAMed	= 2	640 × 200;	2 color	1 page
MCGAHi	= 3	640 × 480;	2 color	1 page
EGALo	= 0	640 × 200;	2 color	4 pages
EGAHi	= 1	640 × 350;	16 color	2 pages
EGA64Lo	= 0	640 × 200;	16 color	1 page
EGA64Hi	= 1	640 × 350;	4 color	1 page
EGAMonoHi	= 3	640 × 350;	2 color	64K: 1 page; 256K: 4 pages
HercMonoHi	= 0	720 × 348;	2 color	2 pages
ATT400C1	= 0	320 × 200;	Red/Yellow/Green	1 page
ATT400C2	= 1	320 × 200;	Cyan/Magenta/White	1 page
ATT400Med	= 2	640 × 200;	2 color	1 page
ATT400Hi	= 3	640 × 400;	2 color	1 page
VGALo	= 0	640 × 200;	16 color	4 pages
VGAMed	= 1	640 × 350;	16 color	2 pages
VGAHi	= 2	640 × 480;	16 color	1 page
VGAHi2	= 3	640 × 480;	2 color	1 page
PC327OHi	= 0	720 × 350;	2 color	1 page
IBM8514Lo	=	640 × 480;	256 colors	1 page (V5.0)
IBM8514Hi	=	1024 × 768;	256 colors	1 page (V5.0)

- The driver that was loaded was found to be faulty somehow.
- There was not enough memory to load the driver.

The error codes supplied by the BGI are always negative integers. The codes are returned in two places: In the **GraphDriver** parameter and by an integer function called **GraphResult**. By testing the **GraphDriver** parameter immediately after a call to **InitGraph** to see if it is negative, you can tell an error code from a valid driver number, which will always be positive. The error codes are the same in both cases, and are summarized in the following table:

Table 22.3
Error Codes Returned by **InitGraph**

Error	*Meaning*
−1	No graphics device is detected in the system
−3	The BGI cannot find the driver file
−4	The driver that was loaded is invalid
−5	There is insufficient memory to load the required driver

The most commonly encountered error is −3, which occurs when the BGI cannot find the requested driver. Recall that the pathname of the driver must be passed in the string parameter **DriverPath**. If **DriverPath** is passed a null string, the BGI will assume that the drivers are in the current directory.

Note in the example invocation of **InitGraph** given on page 465, that the driver pathname is "hard-coded" into the **InitGraph** procedure call. In general, that's not a good thing to do. *You* may always have your drivers stored on drive C: in a subdirectory called BGI, but someone else who may want to run your programs may not.

The preferred method is to assume that the drivers are in the current directory and *try* to load them; if on checking the value returned in **GraphDriver** or **GraphResult** you find an error, bring up a prompt for the user asking him to enter the pathname for the graphics drivers. Once the user enters a pathname, you should re-attempt BGI initialization with a second call to **InitGraph**.

I'm not entirely comfortable with that scheme, as it assumes that the user knows where the BGI drivers are; in reality, the user may not even know what a BGI driver *is*. An interesting project in Turbo Pascal would be to take the general algorithm from the **Locate** utility given in Section 20.9 and build a search routine that will search the current disk drive for files with a .BGI extension, pausing to attempt BGI initialization anywhere it finds one. This would be an ideal application for the directory search "engine" described (for Turbo Pascal 5.0 users) in Section 23.8.

Returning to Text Mode

If your program operates in graphics mode, it can't simply return to DOS when it finishes its work. DOS functions strangely in graphics mode at times; you may get a solid block cursor, but you may get no cursor at all, or you may get a graphics board that simply goes nuts. *Always go back to text mode before returning control to DOS.*

Furthermore, the place to do it is in your program's exit procedure. Why? In case of a runtime error of some sort, the exit procedure will always be executed before control returns to DOS. If you simply code the return to text mode at the normal exit point from the program, a runtime error will leave the user in graphics mode when it aborts your program and returns to DOS.

The procedure that does the job:

```
PROCEDURE CloseGraph;
```

It takes no parameters. When invoked, **CloseGraph** returns to the text mode that was in force when **InitGraph** was executed to enter graphics mode. The dynamic memory that was occupied by the loaded graphics driver is returned to the heap.

The method for invoking **CloseGraph** from within an exit procedure is shown in the **Scribble** program given in Section 22.9.

Moving Between Text and Graphics Modes

The procedures **InitGraph** and **CloseGraph** are the beginning and ending of graphics activity. In between, you may wish to bounce back and forth between text and graphics modes for various reasons. You needn't initialize the BGI from scratch every time you wish to enter graphics mode. Once the BGI has been successfully initialized via **InitGraph**, you can re-enter graphics mode quickly using the **SetGraphMode** procedure:

```
PROCEDURE SetGraphMode(Mode : Integer);
```

The **Mode** parameter must contain a graphics mode code that is valid for the current graphics device. Trying to set an invalid mode will trigger an error code of −10 that will be returned in the **GraphResult** function. The legal modes for the supported graphics devices are given in the table on page 465.

SetGraphMode clears the graphics screen, and, when called from within graphics mode, may be used as a graphics equivalent of the **ClrScr** routine for clearing the text screen.

SetGraphMode is also necessary for changing the graphics mode from the mode the BGI sets when it detects a graphics device in the system. If the detected device supports more than one graphics mode, the BGI will enter the mode providing the highest resolution possible. If for some reason you wish to use a lower-resolution mode with that device, you will have to use the **SetGraphMode** procedure to change to that mode.

Returning to text mode without shutting down the BGI is done with another procedure:

```
PROCEDURE RestoreCRTMode;
```

No parameters are necessary; **RestoreCRTMode** returns to the text mode that was in force when **InitGraph** was first called to initialize the BGI. If the text mode was changed during some previous foray into text mode through **RestoreCRTMode**, the original text mode will be re-asserted.

Using **SetGraphMode** and **RestoreCRTMode** you can alternate between text and graphics modes as many times as you need to. The BGI remains initialized and the current driver remains loaded in memory until **CloseGraph** shuts it down and returns memory occupied by the driver to the heap's free list.

Querying the Current Graphics Mode

The BGI includes a function that will return the current graphics mode:

```
FUNCTION GetGraphmode : Integer;
```

The value returned is one of the mode values summarized in the table on page 465.

Obtaining Names of Graphics Drivers and Modes (Version 5.0)

Version 5.0 adds several new BGI routines for determining the names of the loaded graphics driver and any graphics mode. Obtaining the graphics driver name is done through a simple string function:

```
FUNCTION GetDriverName : String;
```

Obviously, you can't call **GetDriverName** until you successfully complete a call to **InitGraph**. The string name returned is one of the constant identifiers summarized in the table on page 464.

Querying the name of a graphics mode is done through another string function:

```
FUNCTION GetModeName(ModeNum : Word) : String;
```

Here, the number of the mode (from the table on page 465) is passed as a word value, and the string name (see that same table) is returned as the string function result.

Obtaining the maximum mode number for the currently loaded graphics driver is done through the **GetMaxMode** function:

```
FUNCTION GetMaxMode : Word;
```

The result is the highest ordinal mode number supported by that driver, obtained by querying the driver directly.

What are these three functions for? Version 5.0 allows third-party vendors to provide graphics drivers for the BGI, drivers for which neither the BGI nor Borland's documentation may have any knowledge. By and large, within a given driver the highest-numbered mode provides the highest resolution and, by implication, the best quality graphics. Once a driver is loaded, **GetMaxMode** allows you to specify a graphics mode without either specifying an invalid mode or a mode which does not yield the best quality graphics.

22.3: COORDINATES, VIEWPORTS, AND BASIC DRAWING TOOLS

The pixels on a graphics screen are specified by two numbers: An X value indicating their distance from the left margin of the screen, and a Y value indicating their distance from the top of the screen. The two numbers are given as a pair of coordinates, with the X coordinate given first. For example, 17,141 indicate an X value of 17 and a Y value of 141. The upper left corner of the screen is the *origin*, and it is always numbered 0,0.

Because the BGI operates with many different kinds of graphics devices, the largest legal values for X and Y cannot be hard-coded into the BGI or into your programs. Your programs can call two functions that return the largest legal values for X and Y:

```
FUNCTION GetMaxX : Word;

FUNCTION GetMaxY : Word;
```

GetMaxX returns the X coordinate of the leftmost edge of the display screen. **GetMaxY** contains the Y coordinate of the bottommost edge of the display screen. Keep in mind that these boundary numbers are not equal to the resolution of the screen but the resolution reduced by one, since the origin begins at 0. In other words, for a screen with a resolution of 320 × 200, **GetMaxX** will return 319, and **GetMaxY** will return 199 (Figure 22.1).

Since they are ordinary Pascal functions, you can use them within expressions to "generalize" your drawing routines to take varying screen sizes into account. For example, this statement will draw a line around the boundary of the screen, regardless of the screen's resolution:

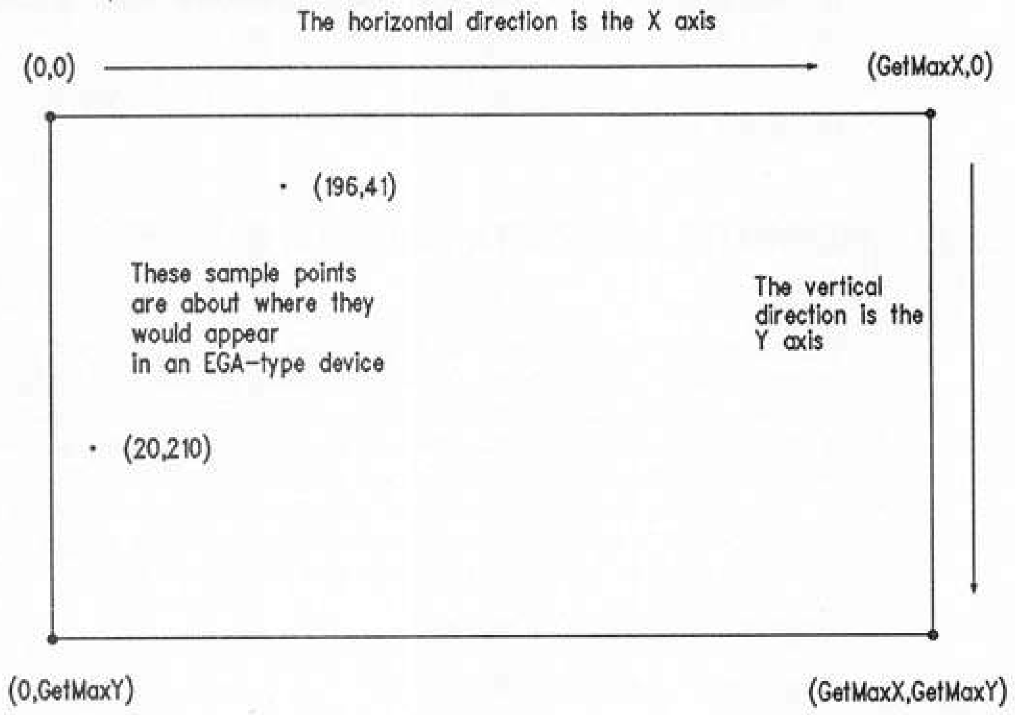
```
Rectangle(0,0,GetMaxX,GetMaxY);
```

Absolute Coordinates

Specifying the coordinates of a pixel on the screen as a specific X and Y value is known as using *absolute coordinates.* In other words, identifying a pixel as 17,244 is a use of absolute coordinates.

Figure 22.1

The Graphics Screen Coordinate System


The horizontal direction is the X axis
(0,0)
(GetMaxX,0)
(196,41)
These sample points are about where they would appear in an EGA-type device
The vertical direction is the Y axis
(20,210)
(0,GetMaxY)
(GetMaxX,GetMaxY)

Some of the drawing procedures in the BGI use absolute coordinates. The best example of this is the routine that simply plots a single pixel at absolute coordinate X,Y:

```
PROCEDURE PutPixel(X,Y : Integer; PixelColor : Word);
```

Here, the absolute coordinates of the position at which you want to plot a pixel are passed as **X** and **Y**. The **PixelColor** parameter is a number specifying the color of the plotted pixel. What values are legal or meaningful here depends heavily on what sort of graphics device you are using. Handling issues of color and line style is described in Section 22.4.

The BGI also includes a routine that plots a straight line between two points, where the points are specified as two pairs of absolute coordinates:

```
PROCEDURE Line(X1,Y1,X2,Y2 : Integer);
```

Here, the two integers **X1** and **Y1** comprise one set of absolute coordinates, and the integers **X2** and **Y2** comprise the other set. The two locations may be anywhere on the screen; that is, the point defined by the first set of coordinates does not need to be above or to the left of that defined by the second set of coordinates.

Line draws its line in the current color and line style, which are set with the **SetColor** and **SetLineStyle** procedures, which will be discussed in detail in Section 22.4.

Two sets of coordinates are sufficient to specify a line, but they are also sufficient to specify a rectangle whose sides are parallel to the sides of the display screen. All you need to do is specify the upper left hand corner of the rectangle, and the lower right hand corner of the rectangle, and it's done. The BGI uses this procedure to do the work:

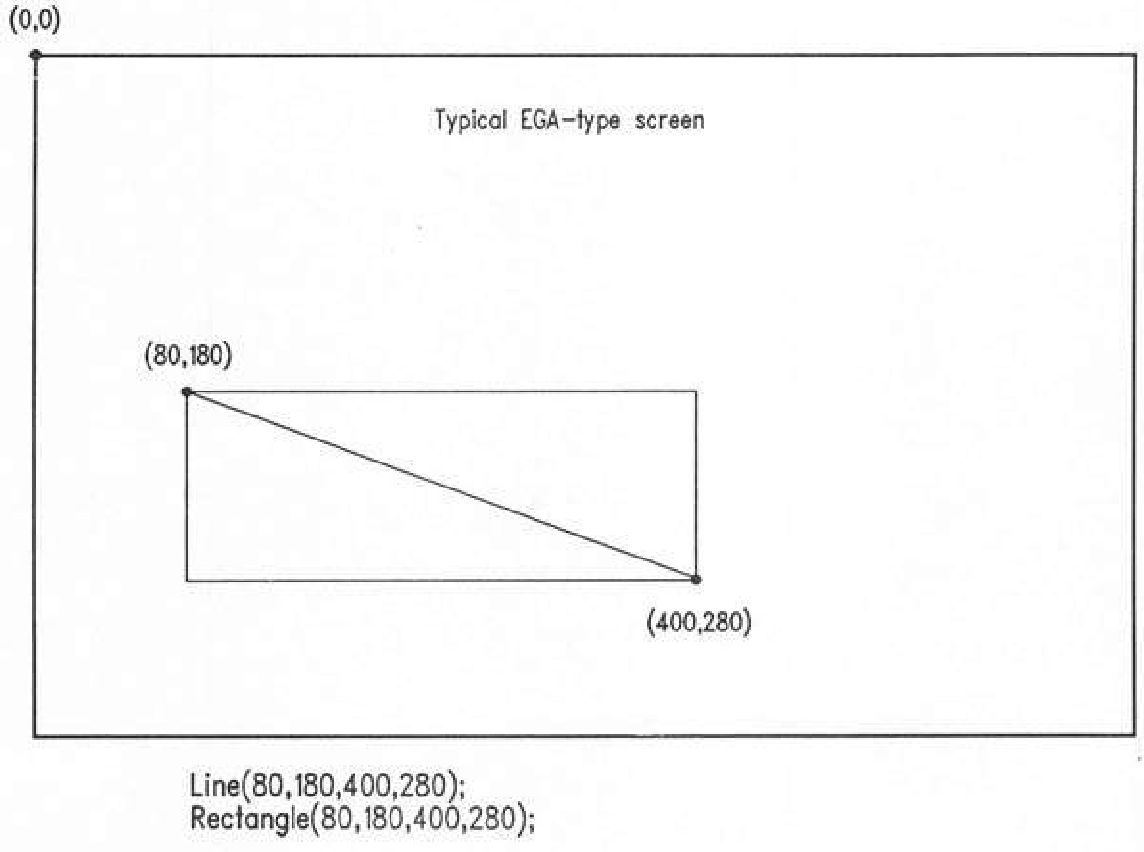
```
PROCEDURE Rectangle(X1,Y1,X2,Y2 : Integer);
```

As with **Line**, the color and style are those currently in force, which may be set with the **SetColor** and **SetLineStyle** procedures, as described in Section 22.4.

The relationship between the output of **Line** and **Rectangle** for the same two sets of absolute coordinates is shown in Figure 22.2. In Figure 22.2, the upper left corner

Figure 22.2

Drawing with Absolute Coordinates



of the rectangle was given in **X1** and **Y1**, but that is not a requirement. The very same rectangle as shown in the figure could have been drawn by reversing the two pairs of coordinates, and specifying the lower-right corner first:

```
Rectangle(400,280,80,180);
```

The Current Pointer (CP)

Absolute coordinates are easy to understand, but in a lot of drawing applications they are more work than they need to be. There are some shorthand techniques offered by the BGI using a more flexible system called *relative coordinates*, and a very handy, if invisible, helper called the *current pointer*. We'll look at the current pointer first.

The current pointer, (usually abbreviated to CP) is the graphics-mode analog to the familiar text cursor. One important difference is that the CP is not visible, as the text cursor is. The CP is actually a pair of coordinates that the BGI keeps within itself as a means of remembering the location of the last pixel drawn to the screen with certain BGI procedures.

When you first initialize the BGI with a call to **InitGraph**, the CP is at 0,0. It represents a starting point for a slightly different method of drawing lines, using a BGI procedure called **LineTo**:

```
PROCEDURE LineTo(X,Y : Integer);
```

Note that **LineTo** only gives you *one* set of coordinates; and to draw a line you need two ends. The other end is provided by the CP. **LineTo** draws a straight line between the CP, wherever it currently happens to be, and the pixel represented by **X,Y**. After the line is drawn, the CP is moved to the other end of the drawn line; in other words, to **X,Y**.

If the CP isn't exactly where you want it to be when you want to draw a line using **LineTo**, you can easily move it with another BGI procedure:

```
PROCEDURE MoveTo(X,Y : Integer);
```

MoveTo does nothing more than move the CP to **X,Y**. Some people find it helpful to think of **MoveTo** as **LineTo** with some mystical "pen" lifted from the screen so that the line is not actually drawn, but I think it is plainer to imagine the CP as an invisible marker that you pick up and carry to a new screen location with **MoveTo**.

The BGI allows you to query the absolute coordinates of the current position of CP. Two functions are provided for this:

```
FUNCTION GetX : Integer;

FUNCTION GetY : Integer;
```

GetX returns the X coordinate of CP, and **GetY** returns the Y coordinate of CP. These two functions return a value that is "viewport-relative," that is, they return the position of CP *relative to the origin of the current viewport.* We'll come back to what that means later in this section when we discuss viewports. Unless you set the current viewport to some subset of the screen, the current viewport is always the entire screen. So, in your early explorations of the BGI, you needn't worry about the consequences of being viewport-relative.

Relative Coordinates

The true power of the current pointer is realized only in conjunction with a different system of specifying locations on the screen, known as *relative coordinates.* Absolute coordinates, to review, may be used to draw a line by naming the ends of the line as specific locations on the screen:

```
Line(0,0,100,280);
```

Relative coordinates do not deal in specific screen locations given as X,Y coordinate pairs. Relative coordinates instead specify a *distance* from some point on the screen, usually CP. Relative coordinates are almost always given as pairs, where the first value in the pair is the distance along the X axis in pixels, and the second value is the distance along the Y axis.

In other words, if you assume your starting point is CP, a relative coordinate pair 5,10 specifies a point 5 pixels to the right of CP and 10 pixels below CP.

These distances in X and Y can be negative as well as positive. A negative X value indicates a position to the *left* of CP, and a negative Y value indicates a position *above* CP. Figure 22.3 shows the difference between absolute and relative coordinates for several points on the graphics screen.

The BGI includes a procedure to move a CP to a new position relative to its current position:

```
PROCEDURE MoveRel(DX,DY : Integer);
```

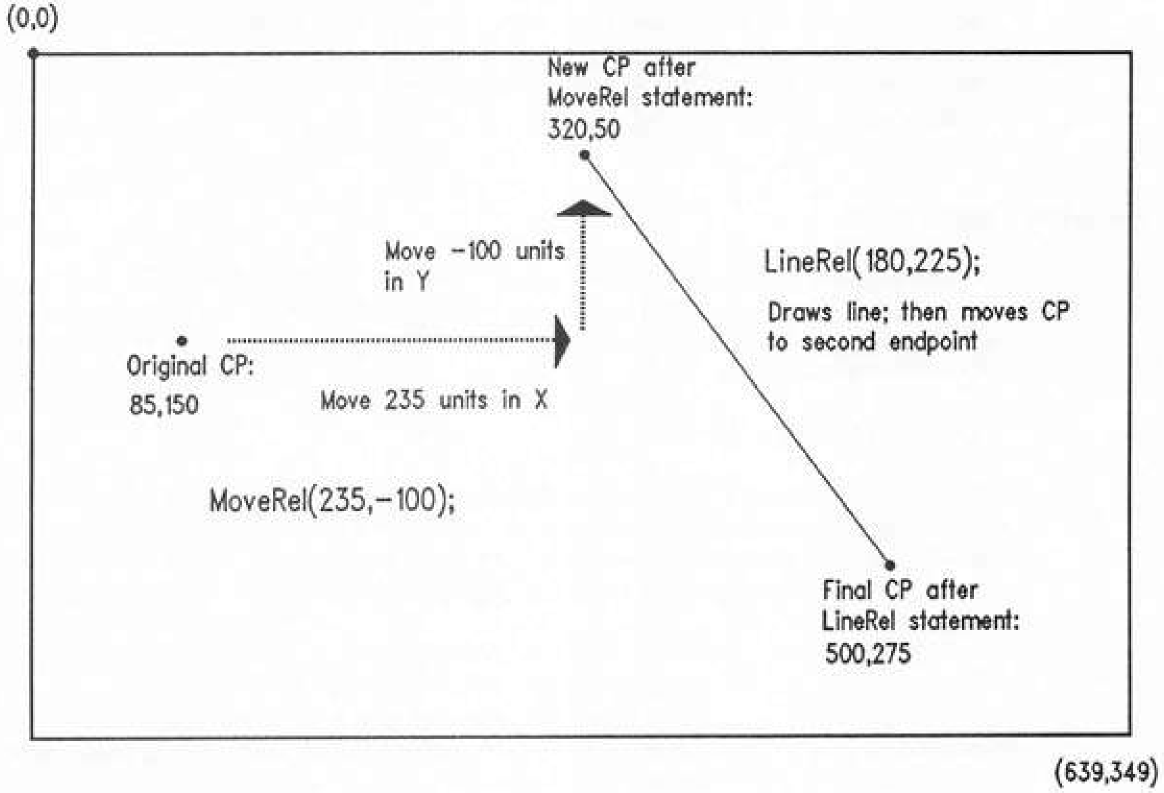
The parameters **DX** and **DY** (named after "delta X" and "delta Y," meaning "change in X" and "change in Y") are the relative coordinates to which CP is moved. For example, if those relative coordinates were 10,−12, the CP would be moved 10 pixels to the right of its current position and 12 pixels above its current position. Keep in mind that a positive distance in X is to the right, while a negative distance in X is to the left; and a positive distance in Y is downward while a negative distance in Y is upward.

The most common use of relative coordinates is probably the drawing of lines from CP to a second position given as relative to CP. The BGI contains a procedure to do this:

```
PROCEDURE LineRel(DX,DY : Integer);
```

Figure 22.3

Drawing with Relative Coordinates



DX and **DY** are, again, relative coordinates. CP is one endpoint of the line to be drawn, and the other endpoint is given relative to CP.

As an example, see Figure 22.3. The first statement:

```
MoveRel(235,-100);
```

moves the CP from its current position at 85,150 to a new position at 85,150 added to the relative coordinates 235,−100, yielding a new absolute position for the CP at 320,50. The second statement:

```
LineRel(180,225);
```

draws a line from CP to the point 180 pixels to the right of CP and 225 pixels below CP. After the line is drawn, **LineRel** moves CP to the second endpoint of the line, at 500,275.

LineRel and **MoveRel** can be used to draw irregular figures specified by a set of points. The sets of points may be stored in an array, and the figure may be drawn by a simple procedure to which the array is passed.

Such a routine, **DrawMarker**, is given below. It draws small figures that are called *polymarkers* in the traditional graphics industry. Polymarkers are small figures used to visibly mark a location on a screen in a distinctive fashion. (The "poly" comes from the fact that the figures are composed of several straight lines and are therefore considered polygons.) The typed constants defined above the procedure itself contain the pairs of relative coordinates used to draw the markers. Three markers are given: A lozenge (i.e., a diamond standing on one point), a cross, and a square.

The first relative coordinate pair in each typed constant is considered a move coordinate by **DrawMarker**, and represents a move *away* from the CP before beginning to draw the marker. If a marker actually begins drawing at the CP (as **Cross** does) this first coordinate pair is 0,0.

After the first move is made, a **WHILE** loop continues drawing lines until a 0,0 relative coordinate is encountered, indicating that the figure is complete.

```
{->>>>DrawMarker<<<<--------------------------------------------}
{                                                               }
{ Filename : DRAWMARK.SRC -- Last Modified 7/10/88              }
{                                                               }
{ This routine uses relative line draws to draw "polymarkers"  }
{ at the current pointer (CP).  The patterns to be drawn are    }
{ arrays of "deltas", each delta being a "change in X" and      }
{ "change in Y" integer pair.  The first pair is a MOVE not a   }
{ line draw, to allow the marker to be drawn entirely away      }
{ from the CP.                                                  }
{                                                               }
{ This routine may only be used in graphics mode.               }
{                                                               }
{                                                               }
{                                                               }
{---------------------------------------------------------------}

TYPE
  PointArray = ARRAY[0..9,0..1] OF Integer;

CONST
  Lozenge : PointArray =
  ((0,-3),(-3,3),(3,3),(3,-3),(-3,-3),(0,0),(0,0),(0,0),(0,0),(0,0));
  Cross   : PointArray =
  ((0,0),(0,-3),(0,6),(0,-3),(3,0),(-6,0),(0,0),(0,0),(0,0),(0,0));
  Square  : PointArray =
  ((-2,-2),(0,4),(4,0),(0,-4),(-4,0),(0,0),(0,0),(0,0),(0,0),(0,0));

PROCEDURE DrawMarker(Marker : PointArray);

VAR
  I : Integer;

BEGIN
  MoveRel(Marker[0,0],Marker[0,1]);  { First pair is relative move }
```

```
    I := 1;    { Start drawing with coordinate pair 1, not 0! }
    WHILE NOT ((Marker[I,0] = 0) AND (Marker[I,1] = 0)) DO
      BEGIN
        LineRel(Marker[I,0],Marker[I,1]);
        Inc(I)
      END
  END;
```

Once drawn, a marker can be erased by setting the current color to the background color and drawing the marker again in the background color.

The markers I've defined here are tiny things, but there is no reason **DrawMark** could not draw any larger figure bounded by eight lines or fewer. Just define the figure in terms of nine points and create a **PointArray** containing the point coordinates relative to the starting point.

Viewports and Viewport-Relative Drawing

The BGI supports *viewports*, which are rectangular windows within your graphics screen. BGI viewports do two things. They *clip*; that is, if you put a viewport in force and draw to the viewport, any graphics that extend beyond the boundaries of the viewport will not be drawn. We say that those "loose ends" are clipped. Also, BGI viewports *translate*. The upper left corner of the viewport becomes position 0,0, and both relative and absolute coordinates are calculated from the upper left corner of the viewport rather than the upper left corner of the entire screen.

Also, you can clear a viewport without disturbing graphics existing outside the boundaries of the viewport.

You should note that viewports do not *scale*; that is, a viewport is not a miniature screen to which all graphics are drawn reduced in size in proportion. A square measuring 50 pixels on a side when drawn on the default viewport (the entire screen) will still be 50 pixels on a side when drawn within a smaller viewport.

When **InitGraph** is executed, the current viewport defaults to the entire screen. The BGI includes a procedure that defines a smaller viewport:

```
PROCEDURE SetViewPort(X1,Y1,X2,Y2 : Integer; Clip : Boolean);
```

The first four parameters contain two sets of coordinates. As with the **Rectangle** procedure, they specify the upper left corner and the lower right corner of the new viewport. The **Clip** parameter is a Boolean value indicating whether or not graphics should be clipped to the new viewport. If **Clip** is passed a value of **True**, then lines or other figures specified as extending beyond the viewport will not be fully drawn. Only those portions of the graphics figures falling within the viewport will be drawn. On the other hand, a value of **False** passed in **Clip** will force all graphics to be drawn regardless of where it falls on the screen.

Keep in mind that, of necessity, the edges of the physical screen are clipping boundaries.

Once a viewport is set, the upper left corner of that viewport becomes the new 0,0 position for graphics. All executed graphics commands will calculate their coordinates relative to that corner. Figure 22.4 shows a simple example. The same sequence of graphics statements is executed twice, once with the current viewport set to the entire screen, and again with the viewport set to a rectangle in the lower right portion of the screen.

```
Rectangle(0,0,GetMaxX,GetMaxY); { Draw a line around the screen }
Circle(50,50,25);               { Draw a thingie }
Line(50,50,100,50);
Line(50,50,50,200);
Line(50,50,100,100);
Rectangle(50,165,100,200);

SetViewPort(320,175,600,300,True);  { Create a viewport }

Rectangle(0,0,GetMaxX,GetMaxY); { Draw a line around the viewport }
Circle(50,50,25);               { Draw the thingie again }
Line(50,50,100,50);
Line(50,50,50,200);
Line(50,50,100,100);
Rectangle(50,165,100,200);
```

Notice that everything is drawn with absolute coordinates, which are the same in both cases, but that the coordinates are *translated* by the viewport to become *viewport-relative*. The graphics figure is drawn the same size in both cases, but its position changes when drawn within the viewport. Also, when drawn within the viewport, the graphics figure loses its bottom half (a square and part of the vertical line) to clipping.

There's one other interesting thing about Figure 22.4: The line supposedly drawn around the viewport is only drawn on two sides of the viewport. The bottom and the right side of the line are clipped. This points up an error in logic that occurs in the graphics statement:

```
Rectangle(0,0,GetMaxX,GetMaxY);
```

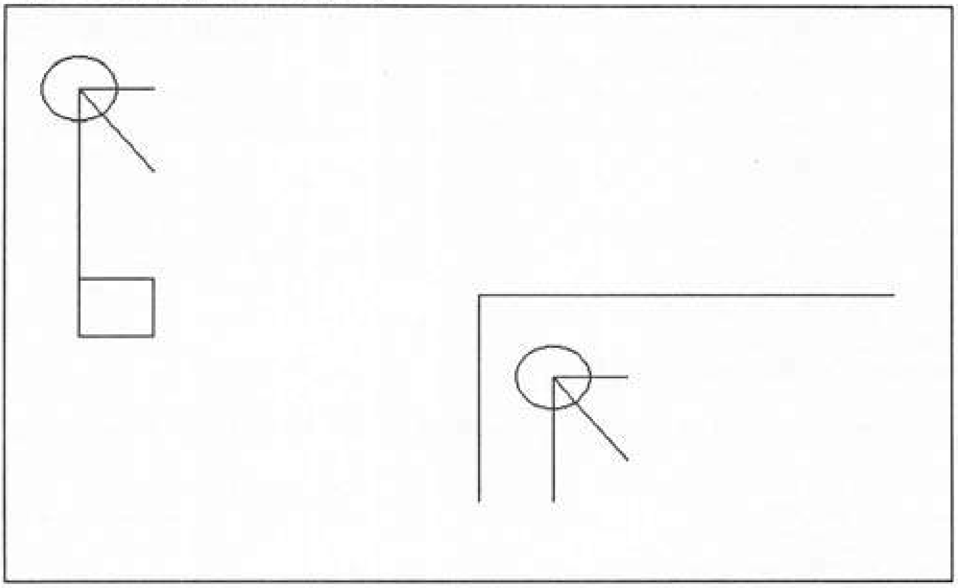
when drawn within the viewport. ***GetMaxX** and **GetMaxY** are not viewport-relative.* They are one of the few features of the BGI that have no truck whatsoever with viewports. **GetMaxX** and **GetMaxY** return the coordinates of the lower right hand corner of the physical screen, no matter what the current viewport happens to be.

Is there any way to query the BGI for the coordinates of the current viewport? Yes indeed:

```
PROCEDURE GetViewSettings(VAR PortSpec : ViewPortType);
```

Figure 22.4

Graphics Clipping with Viewports



When invoked, **GetViewSettings** returns a record value of type **ViewPortType**, which is predefined in the **Graph** unit:

```
ViewPortType = RECORD
                 X1,Y1,X2,Y2 : Word;
                 Clip : Boolean
               END;
```

The proper way to draw a line around the periphery of the current viewport would be this:

```
PROCEDURE OutlineCurrentPort;

VAR
  CurrentPort : ViewPortType;

BEGIN
  GetViewSettings(CurrentPort);
  WITH CurrentPort DO Rectangle(0,0,X2-X1,Y2-Y1);
END;
```

The coordinates passed to **Rectangle** here might seem peculiar. But remember, **Rectangle** is *always* viewport-relative, and **GetViewSettings** returns the original two sets of coordinates that were used to define the current viewport through the **SetViewPort** procedure, which were relative to the *full screen.* What we must do is pass **Rectangle** the coordinates 0,0 as the upper left corner (that's easy enough to understand) and the *difference* of the X and Y values returned by **GetViewSettings** as the lower right corner.

That is something to remember: *The coordinates passed to **SetViewPort** are always relative to the full screen.* If they were not, how else would you reset the viewport to anything larger than the current viewport?

Setting the current viewport back to the full screen, in fact, is done this way:

```
SetViewPort(0,0,GetMaxX,GetMaxY);
```

The interior of a viewport can be erased to the background color in one easy operation, using yet another BGI built-in procedure:

```
PROCEDURE ClearViewPort;
```

This will, of course, erase the line drawn around the periphery of the viewport with the **OutlineCurrentPort** procedure shown on page 478, since the outline is *within* the viewport and is considered part of the viewport.

The Heartbreak of Rectangular Pixels

Little things do count, and little errors can add up to disastrous problems. One of the worst errors committed by IBM in creating the PC architecture lay in creating numerous graphics adapters that displayed rectangular pixels. In fact, all PC-compatible graphics adapters in common use, except for the 8514 and the "480" modes in the VGA and MCGA, display rectangular pixels.

It isn't just a small matter of a few percent. High resolution mode on the CGA produces pixels that are more than *twice* as high as they are wide.

Hey, what's the problem? Pixels are too small to see their shape, anyway, right?

Yes, but put a lot of them in a row on the screen and the discrepancy adds up. For example, execute the following statement in BGI graphics mode:

```
Rectangle(0,0,100,100);
```

This draws a square from the upper left corner of the screen at 0,0 to a point at 100,100. The figure is therefore 100 pixels on a side. It should be a square, right? It is not. Not unless you're using a 640 × 480 pixel mode on a VGA or MCGA, that is. How severely rectangular it is depends on what graphics board you're actually using, and in what mode.

The first and most serious problem with rectangular pixels is that calculations based on geometric principles have to be adjusted to operate correctly in the ill-behaved Cartesian grid represented by your screen. You may think you're drawing a square, but it comes out a rectangle. The proportions of your graphics and diagrams may be misleading, stretched significantly in the vertical and compressed in the horizontal.

The second, compounding problem is that the deviation from squareness varies all over the map, depending on the mode and the graphics device. CGA high resolution has a ratio of horizontal to vertical measure of 5:12; CGA medium resolution is better but still off at 5:6. This ratio of horizontal (X-axis) measure to vertical (Y-axis) measure is called *aspect ratio.* You'll have to deal with it if you expect your graphics to display in a similar fashion across all BGI-supported modes and devices.

Fortunately, the BGI provides a function that returns the aspect ratio at any given time:

```
PROCEDURE GetAspectRatio(VAR XAspect,YAspect : Word);
```

The two parameters return numbers whose ratio reflects the ratio between the width and the height of the screen pixels in the current graphics device and mode. These numbers are calculated internally to the BGI and are used by the BGI to make sure that the figures produced by **Circle**, **Arc**, and **PieSlice** (see Section 22.5) are always displayed as round, regardless of the current graphics device or mode.

The numbers are large positive integers, with the **YAsp** figure typically returned as 10,000, and the **XAsp** figure some other number in proportion to the screen's current aspect ratio. The EGA's 640 × 350 graphics mode, for example, will return a **YAsp** = 10,000 and an **XAsp** = 7750. The ratio of 7750:10,000, or 0.775, accurately reflects the rectangular distortion of that mode.

To draw a true square in any supported mode, you must take aspect ratio into account. The following procedure does all that is necessary:

```
{->>>>Square<<<<------------------------------------------------}
{                                                               }
{ Filename : SQUARE.SRC -- Last Modified 7/23/88                }
{                                                               }
{ This routine draws a square at X,Y that is symmetrical        }
{ independent of the curren graphics device and mode.  The      }
{ Side parameter contains the measure in pixel of a side, and   }
{ the MeasureXAxis is a Boolean that indicates whether the      }
{ figure passed in Side is measured along the X Axis or the Y   }
{ axis.  MeasureXAxis=True assumes that Side measures along     }
{ X axis, and False assumes that Side measures along the Y      }
{ axis.                                                         }
{                                                               }
{ The system must be in BGI graphics mode.                      }
{                                                               }
{                                                               }
{                                                               }
{---------------------------------------------------------------}

PROCEDURE Square(X,Y,Side : Word; MeasureXAxis : Boolean);
```

```

VAR
  XA,YA   : Word;
  XL,YL : Word;

BEGIN
  XL := Side; YL := Side;
  GetAspectRatio(XA,YA);
  IF MeasureXAxis THEN YL := Round((XA/YA)*Side)
    ELSE XL := Round((YA/XA)*Side);
  Rectangle(X,Y,X+XL,Y+YL);
END;
```

Here, we pass procedure **Square** the **X,Y** coordinates of the upper left corner of the square to be drawn, plus the length in pixels of one side. Now, we also need to specify along which axis this measure is to apply, since 100 pixels along the X axis is not as long as 100 pixels along the Y axis. This is the purpose of the **MeasureXAxis** parameter. If set to **True**, **MeasureXAxis** specifies that **Side** is to be measured along the X axis, and the Y length of a side in pixels is to be adjusted according to the aspect ratio. Similarly, if **MeasureXAxis** is set to **False**, then **Side** is assumed to be measured along the Y axis, and the X length in pixels is adjusted according to the aspect ratio. This means you may well get two very different sized squares for two different Boolean values passed in **MeasureXAxis**, particularly if (as in CGA high resolution mode) there is a drastic difference in proportion between the two axes.

Certain things cannot be adjusted by the aspect ratio; specifically, bitmapped things like fill patterns and bit images will look different depending on the device and mode, and there is nothing either you or the BGI can do about it. This is what we pay for a lack of forethought on IBM's part. If Apple ever wins the PC/Macintosh war, it could well be that victory came to them on the back of a point of light almost too small to see.

Fine-Tuning the Aspect Ratio (Version 5.0)

The BGI corrects for rectangular pixels when drawing certain figures, most importantly circles using the **Circle** procedure. Each graphics adapter has a design aspect ratio (for example, 5:6 for the CGA in medium resolution mode), and this is the ratio that the BGI assumes when performing its corrections. The *physical* aspect ratio (that is, the ratio you would get by actually laying a steel rule against the glass of your screen and measuring vertical against horizontal) depends somewhat on the nature of the display. Some lap-held machines have CGA-compatible displays with true square pixels. Or, more commonly, CRT monitors may be misadjusted in the vertical, causing a small but noticeable deviation from the design aspect ratio.

The BGI allows you to override its correct factor, which is the aspect ratio obtained by the **GetAspectRatio** function. The procedure that does this is **SetAspectRatio**:

```
PROCEDURE SetAspectRatio(XAspect,YAspect : Word);
```

This works a lot like **GetAspectRatio** in reverse: We provide values for **XAspect** and **YAspect** and the BGI uses them to calculate its new aspect ratio correction factor, which will be returned in subsequent calls to **GetAspectRatio**.

The two figures can be anything at all, but I recommend holding **XAspect** at 10,000 and altering **YAspect** as necessary. In the case of a lap-held liquid crystal CGA compatible display with square pixels, you would set **YAspect** to 10,000 as well. Or, if your monitor is simply adjusted a little off, you may need a calibration routine that displays a square on the screen and allows you to "squeeze" it with the cursor keys until it looks right, each time changing the aspect ratio slightly. A program to explore aspect ratios in this fashion is given below. Using it, you can adjust the ratio until a square is precisely a square as measured on the faceplate of your CRT. Then you can use the display aspect ratio values as parameters to **SetAspectRatio** in the future to get the ratio just right.

Also, if you would like to build aspect-ratio adjustment into your own graphics programs, you can simply lift the **AdjustAspectRatio** procedure from the program and build it into your own.

```
{--------------------------------------------------------------}
{                         AspectRatio                          }
{                                                              }
{         Aspect ratio adjustment demonstration program        }
{                                                              }
{                            by Jeff Duntemann                 }
{                            Turbo Pascal V5.0                 }
{                            Last update 7/4/88                }
{                                                              }
{                                                              }
{                                                              }
{--------------------------------------------------------------}

PROGRAM AspectRatio;

USES Crt,Graph;

VAR
  I,Color     : Integer;
  Palette     : PaletteType;
  GraphDriver : Integer;
  GraphMode   : Integer;
  ErrorCode   : Integer;


{$I SQUARE.SRC}      { Described in Section 22.3 }


PROCEDURE AdjustAspectRatio;

VAR Side   : Integer;
    W      : Word;
    Ch     : Char;
    Quit   : Boolean;
    Delta  : Integer;
    Color  : Word;
    Filler : FillSettingsType;
    TheLine          : String;
    XAspect,YAspect : Word;


PROCEDURE ShowRatio;

VAR
  Temp : String;

BEGIN
  SetFillStyle(0,0); Bar(0,0,GetMaxX,20);
  WITH Filler DO SetFillStyle(Pattern,Color);
  GetAspectRatio(XAspect,YAspect);
  TheLine := 'Current ratio: ';
  Str(XAspect:6,Temp);
  TheLine := TheLine + Temp + '/';
  Str(YAspect:6,Temp);
  TheLine := TheLine + Temp + '  Arrows to adjust; Q quits...';
  OutTextXY(10,10,TheLine)
END;


BEGIN
  Quit := False; Side := 180;
  Color := GetColor; GetFillSettings(Filler);
  GetAspectRatio(XAspect,YAspect);
  Delta := YAspect DIV 100;
  Square((GetMaxX DIV 2)-(Side DIV 2),
         (GetMaxY DIV 2)-(Side DIV 2),Side,True);

  ShowRatio;
  REPEAT
    Ch := ReadKey;
    IF Ch <> #0 THEN
      IF Ch in ['Q','q'] THEN Quit := True ELSE Quit := False
    ELSE
      BEGIN
        Ch := ReadKey;
        CASE Ord(Ch) OF
          $48 : BEGIN
                  SetColor(0);
                  Square((GetMaxX DIV 2)-(Side DIV 2),
                         (GetMaxY DIV 2)-(Side DIV 2),Side,True);
                  {Kluge fix:} W := YAspect+Delta;
                  SetAspectRatio(XAspect,W);
                  SetColor(Color);
                  Square((GetMaxX DIV 2)-(Side DIV 2),
                         (GetMaxY DIV 2)-(Side DIV 2),Side,True);
                  ShowRatio;
                END;
          $50 : BEGIN
                  SetColor(0);
                  Square((GetMaxX DIV 2)-(Side DIV 2),
                         (GetMaxY DIV 2)-(Side DIV 2),Side,True);
                  {Kluge fix:} W := YAspect-Delta;
                  SetAspectRatio(XAspect,W);
                  SetColor(Color);
                  Square((GetMaxX DIV 2)-(Side DIV 2),
                         (GetMaxY DIV 2)-(Side DIV 2),Side,True);
                  ShowRatio;
                END;
        END; { CASE }
      END
  UNTIL Quit
END;


BEGIN
  GraphDriver := Detect;  { Let the BGI determine what board we're using }
  InitGraph(GraphDriver,GraphMode,'');
  IF ErrorCode <> 0 THEN
    BEGIN
      Write(Chr(7));
      Writeln('>>Halted on graphics error: ',GraphErrorMsg(ErrorCode));
      Halt(2)
    END;

  AdjustAspectRatio;

  CloseGraph;
END.
```

22.4: COLORS, PALETTES, FILLS, AND FLOODS

There are quite a number of routines in the **Graph** unit for setting and querying the use of color in programs that use the BGI. If you're using a monochrome display like the Hercules board, most of them will be unnecessary and you can use the default values. But if you wish to make effective use of color, you should spend a little time understanding the way the BGI throws colors to your screen.

Foreground and Background

In its most commonly encountered form, graphics consists of illuminated pixels on a dark (generally black) background. This is actually a special case of the more general truth: All pixels that are not specifically set to some *foreground color* remain set to some *background color*. Metaphorically, the background may be seen as a sheet of paper on which foreground graphics can be drawn in several colors. There can be many foreground colors (up to 16 in the current release of the BGI, for those graphics devices that support 16 colors) but only *one* background color.

The background color can be changed, and when it is changed, all pixels that belong to the background change to the new color at once.

Colors are specified to the BGI as numbers, but there are 16 predefined constants that help you in remembering which color goes with which number (Table 22.4). Note that not all of these colors are available in every graphics device. The CGA, for example, only supports four colors at once in its **CGAC1** and **CGAC2** modes, and the Hercules board supports none of them at all except for **Black** and **White**.

Setting the background color is done with a single BGI procedure:

```
PROCEDURE SetBkColor(Color : Word);
```

You may specify the **Color** parameter as a variable, a literal, or as one of the constants from the table above. All pixels belonging to the background change to the new color instantly. Furthermore, the background color does not respect viewport boundaries. When you set the background color, you set it for the entire visible screen, *even if a smaller viewport is currently in force.*

Querying the background color is done with a simple BGI function:

```
FUNCTION GetBkColor : Word;
```

The returned value ranges from 0 through 15, depending on what the current graphics device and mode allow.

The drawing routines we have discussed already (**Line**, **LineRel**, and **Rectangle**) draw in the *current drawing color.* This is a color selected from one of the valid colors for the current graphics device, and the drawing routines set pixels to that color as a foreground color. The current drawing color is set with another BGI procedure:

```
PROCEDURE SetColor(Color : Word);
```

As with **SetBkColor**, the parameter **Color** may be passed a variable, a numeric literal, or one of the color constants from the table given above.

Querying the current drawing color is done with a BGI function:

```
FUNCTION GetColor : Word;
```

Table 22.4
Predefined BGI Color Constants

Constant	*Value*	*Constant*	*Value*
Black	0	DarkGray	8
Blue	1	LightBlue	9
Green	2	LightGreen	10
Cyan	3	LightCyan	11
Red	4	LightRed	12
Magenta	5	LightMagenta	13
Brown	6	Yellow	14
LightGray	7	White	15

The value returned will range from 0 through 15, depending on what the current graphics device and mode allow.

The way to erase a figure already drawn in one color is to draw it again, after having temporarily set the current drawing color to the background color:

```
VAR
  SavedColor : Word;

SavedColor := GetColor;       { Record the current drawing color }
Rectangle(0,0,100,100);       { Draw a rectangle in the current color }
SetColor(GetBkColor);         { Set drawing color to background color }
Rectangle(0,0,100,100);       { Draw the rectangle again, erasing it }
SetColor(SavedColor);         { Re-assert the previous drawing color }
```

The number of colors supported by the current driver and mode may be queried with the following function:

```
FUNCTION GetMaxColor : Word;
```

GetMaxColor will return 16 for EGA 640 × 350 mode, and 256 for the VGA 320 × 200 256-color mode. The value returned is the largest value that can be passed to the **SetColor** procedure.

Querying the Color of a Pixel

It may be useful to determine the color of any given pixel on the screen. The BGI provides a function for this purpose:

```
FUNCTION GetPixel(X,Y : Word) : Word;
```

The parameters **X** and **Y** are viewport-relative coordinates of the desired pixel. The result value may range from 0 through 15, depending on what the current graphics device and mode allow.

Palettes

The BGI provides an additional level of versatility in setting display colors, in the form of *palettes.* A palette is, in effect, a translation table for color values. A color value used in drawing by the BGI may be set to any physical color supported by the graphics device with the palette as an intermediary. For example, you can define the BGI's color 3 (ordinarily blue) to be drawn as red by redefining color 3 in the current palette.

Because the palette include the background color as color 0, you can also change the background color by redefining the palette.

Table 22.5 will help make the matter clearer. The left half of the table contains the default palette, in this case for a graphics device like the EGA, VGA, and MCGA, all of which support sixteen simultaneous colors. The CGA, by contrast, only supports four. Notice that each color is "passed through" the table as itself. In other words, color 0 is drawn as color 0; color 1 is drawn as color 1, and so on.

In the right half of the table, the palette has been redefined randomly, so that every color is redefined as some other color. In other words, once the new palette is put into effect, color 0 will be drawn as color 14; color 1 will be drawn as color 4, and so on. All colors do not have to be changed; you may redefine only one, or redefine all but one, or any combination. In the redefined table, BGI color 11 was left defined as physical color 11, **LightCyan**.

Physically, the palette is a table kept somewhere in memory by the BGI. When you alter the current palette, you change values in this table. There are two BGI routines that perform this palette redefinition. One of them redefines one color at a time, and the other allows you to redefine all of them at once. Redefining a single color is done with this procedure:

```
PROCEDURE SetPalette(DrawColor : Word; PhysicalColor : ShortInt);
```

The meaning of the two parameters is something you should keep in mind: **DrawColor** is the value you would use in setting a current drawing color through the **SetColor**

Table 22.5
Default and Custom Palettes

Default Palette		**Redefined Palette**	
BGI Color	*Drawn as*	*BGI Color*	*Drawn as*
0	0 (Black)	0	14 (Yellow)
1	1 (Blue)	1	4 (Red)
2	2 (Green)	2	10 (LightGreen)
3	3 (Cyan)	3	1 (Blue)
4	4 (Red)	4	9 (LightBlue)
5	5 (Magenta)	5	6 (Brown)
6	6 (Brown)	6	12 (LightRed)
7	7 (LightGray)	7	15 (White)
8	8 (DarkGray)	8	13 (LightMagenta)
9	9 (LightBlue)	9	0 (Black)
10	10 (LightGreen)	10	3 (Cyan)
11	11 (LightCyan)	11	11 (LightCyan)
12	12 (LightRed)	12	7 (LightGray)
13	13 (LightMagenta)	13	8 (DarkGray)
14	14 (Yellow)	14	5 (Magenta)
15	15 (White)	15	2 (Green)

procedure. **PhysicalColor** is the actual color in which that current drawing color would appear on the screen.

For example, if you wished color 4 (which defaults to red) to specify light blue on the screen, you would execute this statement:

```
SetPalette(4,LightBlue);
```

Setting a value for all colors in the palette may be done with a different procedure:

```
PROCEDURE SetAllPalette(VAR Palette);
```

As you can tell from the presence of an untyped **VAR** parameter here, the BGI is hedging its bets about the actual physical representation of its internal palette translation table. By refusing to build a specific structure into the BGI procedure definition, Borland has the ability to rearrange the table to some degree in a future release without changing the definition of the **SetAllPalette** procedure.

For the time being, the structure of the palette table is a record as shown below:

```
CONST
  MaxColors = 15;

TYPE
  PaletteType = RECORD
                  TableSize : Byte;
                  Colors    : ARRAY[0..MaxColors] OF ShortInt
                END;
```

The constant **MaxColors** is predefined in the **Graph** unit, and for the current release is limited to 16 colors. Version 5.0 supports more simultaneous colors and a larger palette, particularly under the 256-color modes present in the IBM VGA, 8514, and MCGA graphics devices.

The **TableSize** field in the **PaletteType** record specifies the number of bytes occupied by the translation table. For the time being, that will be equivalent to **MaxColors** plus one, because **MaxColors** counts from 0.

To use **SetAllPalette**, define a variable of type **PaletteType** and assign your translation values to the various fields as needed. Then invoke **SetAllPalette** with your **PaletteType** variable as the parameter. You may assign a value of −1 to any of the elements of the **Colors** array (this is the reason the values are of type **ShortInt**, which is a signed, 1-byte integer) and the −1 value indicates that the previous value in the BGI's table for that color is not to be changed.

Something to keep in mind is that color 0 in the palette table specifies the background color. Unless you wish the background color to be something other than black, don't redefine color 0.

Another thing to remember is that changing the palette affects all graphics information *currently* on the screen, as well as graphics figures drawn *after* the palette is changed. When you change any color in the palette, any graphics drawn in that color anywhere on the screen change *instantly*.

SetAllPalette allows you to change the palette temporarily and then restore the original palette. To do this requires a little help from another BGI procedure:

```
PROCEDURE GetPalette(VAR CurrentPalette : PaletteType);
```

You can use **GetPalette** to return the current palette translation table in the parameter **CurrentPalette**, which is a record of the **PaletteType** type defined above. With the current palette saved in a holding variable of type **PaletteType**, you can change the palette to your special temporary palette, do what work needs to be done, and then change the palette back to the saved palette:

```
VAR
  PaletteStash : PaletteType;

  GetPalette(PaletteStash);
  SetAllPalette(NewPalette);
  DrawSomething;
  SetAllPalette(PaletteStash);
```

As an example of how palettes work, I offer the following short program. It draws a large number of lines in random positions in random foreground colors. Then it alters the palette randomly, and each line drawn in a given color *instantly* changes to the new color. The lines are not being redrawn; the information on the screen is simply being reinterpreted within the new color translation scheme.

I call the program **PsychedelicFiberglas**, which might seem odd at first; but once you compile and run the program, you will see immediately how appropriate a name it is. Especially if, like me, you came of age in the Sixties . . .

```
{--------------------------------------------------------------}
{                    PsychedelicFiberglas                      }
{                                                              }
{                              by Jeff Duntemann               }
{                              Turbo Pascal V5.0               }
{                              Last update 7/13/88             }
{                                                              }
{ This program demonstrates palette-switching through the BGI. }
{ Children of the Sixties will understand the name.            }
{                                                              }
{                                                              }
{                                                              }
{--------------------------------------------------------------}

PROGRAM PsychedelicFiberglas;

USES Crt,Graph;

VAR
```

```
    I,Color     : Integer;
    Palette     : PaletteType;
    GraphDriver : Integer;
    GraphMode   : Integer;
    ErrorCode   : Integer;

  BEGIN
    GraphDriver := Detect;  { Let the BGI determine what board we're using }
    InitGraph(GraphDriver,GraphMode,'');
    IF ErrorCode <> 0 THEN
      BEGIN
        Writeln('>>Halted on graphics error: ',GraphErrorMsg(ErrorCode));
        Halt(2)
      END;

    Randomize;

    GetPalette(Palette);
    FOR Color := 0 TO 10000 DO
      BEGIN
        SetColor(Random(Palette.Size));
        Line(Random(GetMaxX),Random(GetMaxY),Random(GetMaxX),Random(GetMaxY));
      END;

    REPEAT
      REPEAT I := Random(Palette.Size) UNTIL I <> 0;
      SetPalette(I,Random(Palette.Size));
    UNTIL KeyPressed;

    CloseGraph;

  END.
```

Version 5.0 Palette Enhancements

The 5.0 expansion of the maximum-size palette to 256 colors, and the addition of the IBM 8514 display device made palette management more complex than it was under 4.0. New routines were added to the BGI for Version 5.0 to deal with palette complexities.

The palette color lookup table varies in size depending on which graphics driver is loaded. A new function provides the preferred means of determining how many colors are in a particular driver's palette:

```
FUNCTION GetPaletteSize : Word;
```

The return value tells you how many colors are in the current driver's palette. This system replaces the 4.0 method, in which the **GetMaxColor** was the means used to query palette size.

The 8514 and VGA in 256-color mode presents a special problem for palette management, because they have a palette of 256 colors that may be chosen from among a group of 2^{18}, or 256,144. This massive number consists of three 6-bit values, one for Red, one for Green, and one for Blue. The 256-color palette comes filled with the standard default colors, the first 16 of which match the 16 standard colors used with the EGA and VGA high resolution modes. If you wish to experiment with some of those other 256,000-odd colors, you'll need to set them with a new, 5.0-specific procedure:

```
PROCEDURE SetRGBPalette(ColorNum,RedVal,GreenVal,BlueVal : Word);
```

Here, **ColorNum** specifies a number from 0-255 which is the entry in the 256-color palette you wish to specify, and the three remaining parameters provide the color values to be summed up as that entry in the table. Think of it as a means to specify a single 18-bit color number by breaking it down into component colors, each of which in turn are 6-bit intensity values for the red, green, and blue electron guns in an RGB analog monitor.

Line Styles

We tend to think of lines as mundane, from-here-to-there strokes of a single, uninterrupted color. In a simpler graphics system (such as that found in Turbo Pascal V3.0), this was the case. The BGI, however, allows you to add a little life to your lines, in two ways: thickness and "style," that is, lines that are drawn in a pattern of dots or dashes or combinations of both.

True, there are only two choices for line width: thin (one pixel wide) and thick (three pixels wide). But the combination of two line thicknesses plus an almost unlimited variety of line styles (assuming you roll your own) provides a lot of latitude in laying down the line.

Line styles affect the patterns drawn by all of the following BGI draw routines: **Line**, **LineRel**, **Rectangle**, and **DrawPoly**. The procedures that draw curved lines (**Circle**, **Arc**, and **Ellipse**) are not affected, nor are the procedures that draw the stroke fonts.

Ordinarily, lines are drawn in the narrow width, without any specific line style. To change either line thickness, style, or both, you must use this BGI procedure:

```
PROCEDURE SetLineStyle(LineStyle : Word;
                       BitPattern : Word;
                       Thickness : Word);
```

There are a number of predefined constants to make your invocations of **SetLineStyle** more meaningful to the casual reader (including yourself, six months after the fact) Table 22.6 summarizes the constants and what they mean.

All three of **SetLineStyle**'s parameters are type **Word**. The **LineStyle** parameter specifies either one of the BGI's four built-in line styles, or else a programmer-defined line style specified by the bit pattern in the second parameter, **BitPattern**. When using one of the four built-in line styles (**SolidLn**, **DottedLn**, **CenterLn**, or **DashedLn**) the

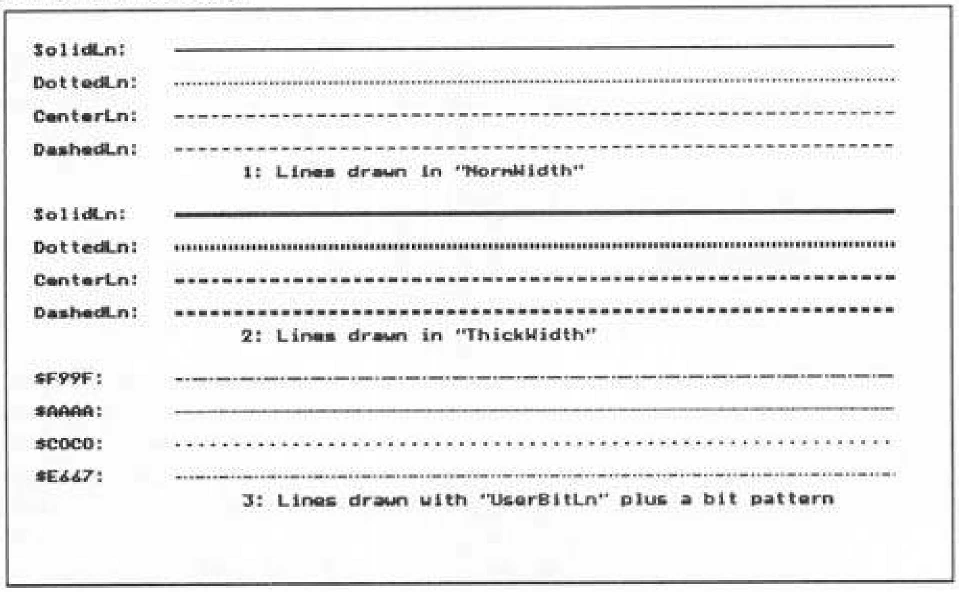
Table 22.6
Predefined LineStyle Constants

Constant	*Value*	*Pass in Parameter*	*Comments*
SolidLn	0	LineStyle	Solid line
DottedLn	1	LineStyle	Fairly coarse dotted line
CenterLn	2	LineStyle	Long dash/short dash pattern
DashedLn	3	LineStyle	Dashed line
UserBitLn	4	LineStyle	Causes BGI to use BitPattern
NormWidth	1	Thickness	1-pixel-wide lines
ThickWidth	3	Thickness	3-pixel-wide lines

BitPattern parameter should be passed a 0. The third parameter, **Thickness**, specifies one of the two BGI-supported line thicknesses. The only two legal values are 1 (**NormWidth**) or 3 (**ThickWidth**). Any other value passed in **Thickness** will be ignored, and lines will be drawn according to the last legal value passed in **Thickness**, or else the default (**NormWidth**) if **SetLineStyle** has not yet been called.

The best way to illustrate what **SetLineStyle** can do is by showing you an actual screen containing various types of lines drawn by the BGI. Figure 22.5 is such a screen.

Figure 22.5
Lines and Line Styles


SolidLn:
DottedLn:
CenterLn:
DashedLn:
1: Lines drawn in "NormWidth"
SolidLn:
DottedLn:
CenterLn:
DashedLn:
2: Lines drawn in "ThickWidth"
$F99F:
$AAAA:
$C0C0:
$E667:
3: Lines drawn with "UserBitLn" plus a bit pattern

The **SetLineStyle** invocations that specified each of the 12 lines displayed in Figure 22.5 are given below, in the same order that the lines appear on the screen:

```
SetLineStyle(SolidLn,0,NormWidth);  { This is the default! }
SetLineStyle(DottedLn,0,NormWidth);
SetLineStyle(CenterLn,0,NormWidth);
SetLineStyle(DashedLn,0,NormWidth);

SetLineStyle(SolidLn,0,ThickWidth);
SetLineStyle(DottedLn,0,ThickWidth);
SetLineStyle(CenterLn,0,ThickWidth);
SetLineStyle(DashedLn,0,ThickWidth);

SetLineStyle(UserBitLn,$F99F,NormWidth);
SetLineStyle(UserBitLn,$AAAA,NormWidth);
SetLineStyle(UserBitLn,$C0C0,NormWidth);
SetLineStyle(UserBitLn,$E667,NormWidth);
```

The first **SetLineStyle** invocation given above is the default condition for line styles, and the values shown will be in force until you change them with a different set of values passed in **SetLineStyle**.

Custom line styles are represented by 16 bits, where a 1 bit will be drawn to the screen in the foreground color, and a 0 bit will be left in the background color. Designing custom bit patterns for your own line styles is as much a matter of trial-and-error as it is planning. Sketch out the line pattern you want to draw on graph paper, blackening in the bits for the foreground color portions of the pattern, Then derive the binary bit pattern by translating the blackened blocks into 1 bits, and the empty blocks into 0 bits. Then display a line using your new line style, and if it isn't quite right, tweak it and try again. Remember always that the pattern "wraps" after 16 bits, and that bit 15 is right next to bit 0. There is no dead space left between them.

One final caution on line styles: The "gaps" in a styled line do not assert themselves when drawn over foreground color information. In other words, a styled line when drawn over a solid line or a filled bar will appear solid. The gaps in the line are *allowed to remain* in the background color, but they are not *forced* to the background color!

Setting the Write Mode for Line Drawing (Version 5.0)

Turbo Pascal 5.0 adds a *write mode* feature to BGI line drawing. A write mode is the logical operation invoked when each pixel comprising the line being drawn is written to the screen. There are currently two write modes, present as named constants in the **Graph** unit:

CopyPut With a value of 0. Here, pixels comprising the line are written directly to the screen, overwriting any pixels underneath the line on the screen. This is the only mode in which lines are written under Turbo Pascal 4.0.

XORPut With a value of 1. When this write mode is in force, the pixels comprising the line are combined with pixels underneath the line on the screen, using the XOR logical operator (see Section 11.4). One interesting and useful property of the XOR logical operator is that a line written to the screen using write mode **XORPut** can be erased by simply writing the same line to the same place on the screen, again with write mode **XORPut** in force.

Setting the write mode is done with a new 5.0-specific procedure:

```
PROCEDURE SetWriteMode(WriteMode : Integer);
```

Here, **WriteMode** may have a value of 0 or 1. The default write mode is **CopyPut**.

Fill Patterns

What line styles are to lines, fill styles are to areas. Actually, there are two ways of filling areas on the screen. One, fill styles, we will discuss here. The other, flood filling, will be discussed in the following section.

Apple's Macintosh computer made fill styles popular. The BGI includes a number of relatively ordinary fill patterns, but it also gives you the option of defining your own. With a little cleverness you can fill an area with a pattern that looks like bricks, or shingles, or staring eyeballs.

Filling an area requires that we specify a fill style and a fill color. There are two different procedures to do this: **SetFillStyle** selects one of the predefined fill patterns; and **SetFillPattern** sets the current fill pattern to a custom pattern defined in an array of 8 bytes. In either case, the style is the pattern that fills the specified area, and the color is the color in which the pattern is drawn.

To choose one of the predefined patterns and an associated color, use **SetFillStyle**:

```
PROCEDURE SetFillStyle(Pattern : Word; Color : Word);
```

The **Color** parameter takes the number of a color that is legal for the current graphics device and mode. Currently, it cannot be larger than 15 for any supported graphics device.

In the **Pattern** parameter you pass one of several legal codes corresponding to predefined BGI fill patterns. As with most of the numerous codes valid for BGI operations, the **Graph** unit contains a number of predefined constants to make remembering the fill pattern codes easier (Table 22.7).

The best way to describe these patterns is simply to show them to you. Figure 22.6

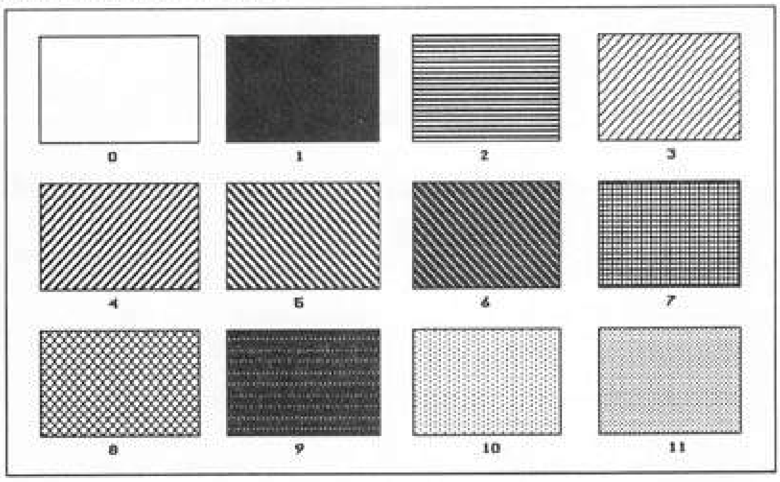
Table 22.7
Predefined Fill Pattern Constants

Constant	*Value*	*Meaning*
EmptyFill	0	Fills with background color
SolidFill	1	Fills with solid foreground color
LineFill	2	Horizontal line fill
LtSlashFill	3	Light forward slant line fill
SlashFill	4	Heavy forward slant line fill
BkSlashFill	5	Heavy backward slant line fill
LtBkSlashFill	6	Light backward slant line fill
HatchFill	7	Light crosshatch fill
XHatchFill	8	Heavy crosshatch fill
InterleaveFill	9	Interleaved line fill
WideDotFill	10	Wide-spaced dot fill
CloseDotFill	11	Narrow-spaced dot fill

is an actual screen dump of all 12 predefined fill patterns supplied with the BGI. The default fill pattern (that is, the fill pattern that is in force until you explicitly set another one with **SetFillStyle** or **SetFillPattern**) is a solid fill with white foreground pixels.

The fill pattern that you select using **SetFillPattern** is the pattern that the BGI will use when you draw filled graphics figures using the **Bar**, **Bar3D**, **FillPoly**, and **PieSlice** procedures.

Figure 22.6
The Predefined BGI Fill Patterns



Defining Your Own Fill Patterns

The BGI provides 12 predefined fill patterns, but you aren't stuck with only those 12. Another BGI procedure allows you to define a custom fill pattern of your own and make it the current fill pattern:

```
PROCEDURE SetFillPattern(Pattern : FillPatternType; Color :

Word);
```

The **FillPatternType** is defined in unit **Graph** and is nothing more than an array of 8 bytes:

```
TYPE
  FillPatternType : ARRAY[1..8] OF Byte;
```

Think of the 8 bytes in **FillPatternType** as a bitmap that is endlessly replicated throughout the area being filled. A 1 bit in any of the 8 bytes becomes an illuminated pixel within the pattern. A 0 bit does not "erase" what is underneath it; rather, you should think of 0 bit portions of the fill pattern as being transparent and will allow any graphics beneath them to "shine through."

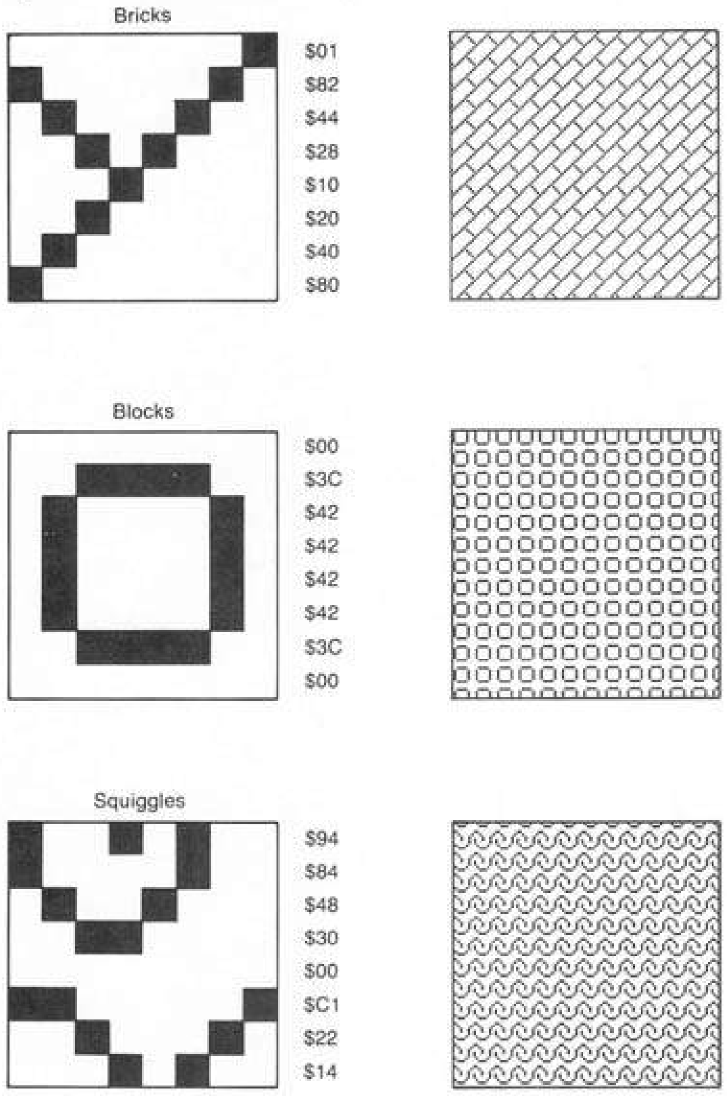
Designing a fill pattern, as with line styles, is best begun on square-ruled graph paper. Lay out an 8 × 8 pixel square area, and darken in the pixels that you wish to be illuminated in your pattern. Translate the pattern of darkened and white blocks into binary bytes, where the darkened squares are 1 bits and the white squares 0 bits. The top row of the design square becomes element 1 of a **FillPatternType** array, the second row of the design square becomes element 2 of the array, and so on. It takes some practice, because you tend to forget that the edges of the pattern join up with themselves. The block you lay out may bear little resemblance to the final pattern it produces in a filled region.

Figure 22.7 shows three such design squares and their translations into binary bytes, along with filled rectangles suggestive of how such patterns appear on an actual screen. Note especially the nonintuitive connection between the design square for the "squiggles" pattern and the pattern itself. It took some considerable trial-and-error to get "squiggles" right, because the wrap from top edge to bottom edge is crucial to the "squiggleness" of the pattern. "Blocks," by contrast, is a self-contained little pattern, its screen appearance obvious from the design square.

I generally store my custom fill patterns as array constants, which is a compact and easily readable way to represent them. The short program given below, PATTERNS.PAS, demonstrates how to store such patterns as array constants, assert them via **SetFillPattern**, and then use them to fill rectangles with the **Bar** procedure. **Patterns** simply draws six rectangles and fills them with six custom fill patterns in white. It is reasonably device-independent and should run on most any BGI-supported graphics device.

Figure 22.7

Pattern Design



```
{--------------------------------------------------------------}
{                          Patterns                            }
{                                                              }
{            Graphics pattern demonstration program           }
{                                                              }
{                             by Jeff Duntemann                }
{                             Turbo Pascal V5.0                }
{                             Last update 7/14/88              }
{                                                              }
{                                                              }
{                                                              }
{--------------------------------------------------------------}

PROGRAM Patterns;

USES Graph;

CONST
  Halftone1 : FillPatternType =
              ($CC,$33,$CC,$33,$CC,$33,$CC,$33);
  Halftone2 : FillPatternType =
              ($AA,$55,$AA,$55,$AA,$55,$AA,$55);
  Squiggles : FillPatternType =
              ($94,$84,$48,$30,$00,$c1,$22,$14);
  Vertical  : FillPatternType =
              ($CC,$CC,$CC,$CC,$CC,$CC,$CC,$CC);
  Bricks    : FillPatternType =
              ($01,$82,$44,$28,$10,$20,$40,$80);
  Blocks    : FillPatternType =
              ($00,$3C,$42,$42,$42,$42,$3C,$00);

VAR
  GraphDriver : Integer;
  GraphMode   : Integer;
  ErrorCode   : Integer;


BEGIN

  GraphDriver := Detect;  { Let the BGI determine what board we're using }
  InitGraph(GraphDriver,GraphMode,'');
  IF ErrorCode <> 0 THEN
    BEGIN
      Writeln('>>Halted on graphics error: ',GraphErrorMsg(ErrorCode));
      Halt(2)
    END;

  SetFillPattern(Halftone1,White);
  Bar(0,0,99,100);
  Rectangle(0,0,99,100);

  SetFillPattern(Halftone2,White);
  Bar(110,0,209,100);
  Rectangle(110,0,209,100);

  SetFillPattern(Squiggles,White);
  Bar(220,0,319,100);
  Rectangle(220,0,319,100);
```

```

    SetFillPattern(Vertical,White);
    Bar(0,105,99,199);
    Rectangle(0,105,99,199);

    SetFillPattern(Bricks,White);
    Bar(110,105,209,199);
    Rectangle(110,105,209,199);

    SetFillPattern(Blocks,White);
    Bar(220,105,319,199);
    Rectangle(220,105,319,199);

    Readln;
    CloseGraph;
  END.
```

Querying Fill Patterns and Colors

As with nearly all graphics attributes supported by the BGI, the fill patterns and colors may be queried by your programs. The routine that does this is:

```
PROCEDURE GetFillSettings(VAR FillData : FillSettingsType);
```

The settings are returned in a record of type **FillSettingsType**, which is predefined within the **Graph** unit:

```
TYPE
  FillSettingsType = RECORD
                       Pattern : Word;
                       Color   : Word
                     END;
```

If you're sharp you'll notice a weakness here. There is only 1 byte in which to report the fill pattern. What **GetFillSettings** reports is only the number of the predefined fill pattern that is currently in force. These numbers are given in the table on page 495. If a custom fill pattern is in force, **Pattern** will contain the code 12. The **Color** field will contain the color value passed to the BGI in **SetFillStyle** or **SetFillPattern**.

Flood Filling

Regular, predictable figures such as rectangles, polygons, and pie slices can be filled using a technique called *scan conversion,* which is fast but limited to that sort of figure whose boundaries are mathematically predictable. Filling a region with irregular or unpredictable boundaries requires another sort of fill technique, called *flood filling*. In simplest terms, when the BGI does a flood fill, it picks a point on the screen at which to begin, and then propagates the pattern away from that point, stopping only when it reaches a boundary of a specified color. The procedure to do it is this:

```
PROCEDURE FloodFill(StartX,StartY : Integer; BoundaryColor :
Word);
```

The parameters **StartX** and **StartY** specify a point in the viewport called a *seed*, which is where the flood operation begins. Flooding is limited by the boundaries of the current clipping viewport, the borders of the physical screen, or a region of foreground color specified by **BoundaryColor**.

The pattern used by **FloodFill** is the current fill pattern, set by either **SetFillStyle** or **SetFillPattern**, as described on page 496. You will notice that flood-filling is considerably slower than filling by scan conversion. There is a lot more testing to do; the BGI must literally test every graphics byte in the display buffer before it fills it, to be sure it doesn't try to fill over or beyond a proper color boundary.

One caution in using **FloodFill** is to be sure the region you intend to fill is truly closed. In other words, the boundary of foreground color specified by **BoundaryColor** must be complete and unbroken, all the way around. Even a 1-pixel gap will let the fill pattern or color "leak out" and spread to fill the entire remaining empty area of the screen or viewport.

In general, use **Bar**, **Bar3D**, **FillPoly**, or **PieSlice** to generate filled areas whenever possible. Not only are they faster methods to fill space with color or a pattern, they appear to be more in line with Borland's future plans for the evolution of the BGI. Particularly, if Borland ever intends to make the BGI a true "world coordinate" graphics system, then flood filling becomes extremely difficult and may be eliminated altogether. The rift has already begun with Turbo Pascal 5.0, since the 8514 driver does *not* support flood filling.

22.5: POLYGONS, BARS, ARCS, CIRCLES, AND ELLIPSES

We've discussed the basic BGI routines for drawing lines and rectangles, and used the **Bar** routine in the previous section to demonstrate the BGI's fill patterns. In this section we'll talk about the other BGI routines that actually draw figures on your screen.

Drawing Generic Polygons

In computer graphics terms, a *polygon* is any figure composed only of straight lines. We're used to thinking of polygons as *closed* figures, that is, where one endpoint comes around and meets the other endpoint, forming an enclosed region. In fact, a wandering line composed of some number of line segments is just as genuinely a polygon, though it is variously called an *open polygon* or a *hull*. The easiest way to define such a figure (given that nothing requires it to be a *regular* polygon) is by defining a set of coordinate points acting as its vertices, and drawing lines from vertex to vertex until the figure is complete. This is the way the BGI operates, through this procedure:

```
PROCEDURE DrawPoly(PointCount : Word; VAR PointSet);
```

Here, **PointCount** specifies the number of vertices the polygon will have. One quirk here is that the beginning and ending vertex are considered separate points and must have separate coordinates, even if they are at an identical position on the screen. This is necessary, again, because in an open polygon the two end vertices *are* separate and distinct points. Obviously, the number of vertices will vary from call to call, so passing the coordinates of the vertices is a bit of a problem. The problem is solved by making the **PointSet** parameter untyped, so that any data type at all can be passed through it.

Actually, for **DrawPoly** to work correctly, what the BGI expects in **PointSet** is an array of a record type called **PointType**. The definition of **PointType** is in unit **Graph**:

```
TYPE
  PointType = RECORD
                X,Y : Integer
              END;
```

You can create arrays of **PointType** as array constants, or build the arrays on the heap, or simply make them global and fill them using assignment statements. Something important to keep in mind: *The vertices of a polygon are specified as absolute and not relative coordinates.* This is a shame, in some ways, because to relocate a polygon to another position on the screen you must recalculate and change *all* values in *every* **PointType** record that specifies a vertex. For a complicated polygon this could take a significant amount of time, but for now it can't be helped.

Below is an array constant that contains the vertex definitions for a tolerable, if not entirely regular, pentagon:

```
Pentagon   : ARRAY[0..5] OF PointType =
             ((X : 100; Y :  10),
              (X : 200; Y :  80),
              (X : 155; Y : 160),
              (X :  45; Y : 160),
              (X :   0; Y :  80),
              (X : 100; Y :  10));
```

To define a polygon with more or fewer vertices, simply extend or reduce the number of elements in the array specification at the top, and fill in or trim the definition accordingly. Drawing the pentagon would be done this way:

```
DrawPoly(6,Pentagon);
```

Remember that the **PointCount** parameter takes the number of vertices in **PointSet**, *not* the high index of the array. Also remember that for closed polygons the number of vertices is one more than we might from common sense expect. The polygon is drawn in the current line style and color, as set by **SetLineStyle**.

The BGI has machinery for drawing filled polygons in a very similar fashion, using a separate procedure:

```
PROCEDURE FillPoly(PointCount : Word; VAR PointSet);
```

In terms of its parameters and its calling syntax and conventions, **DrawPoly** and **FillPoly** are absolutely identical. Their only difference is what they draw on the screen. Both draw the outline of the polygon in the current line style and color, but **FillPoly** also fills the polygon with the current fill pattern. As with **DrawPoly**, the perimeter of the filled polygon is drawn in the current line style and color, as set by **SetLineStyle**.

Ahh, but you may think of the catch: What about open polygons? Common sense tells us that you can't fill an open polygon, which may be no more than a jagged line, and common sense is correct. If you pass an open polygon to **FillPoly**, the BGI will draw the shortest line between the two end vertices and *then* fill it. No error will be generated. In a sense, the BGI guarantees that **FillPoly** will always work with a closed polygon by closing any open polygon that happens to be passed to **FillPoly**.

The short program below defines two polygons, one open, and one closed. The closed polygon is the pentagon defined above, and the open polygon is a crooked line with some passing resemblance to the Big Dipper. POLYGON.PAS demonstrates how polygons are defined in typed constants, and shows how the BGI closes an open polygon before it fills the open polygon with **FillPoly**.

The program also demonstrates one easy method for moving a polygon: Using a **FOR** loop to step through either the X or Y coordinates and adding or subtracting the same value to all coordinates. POLYGON.PAS only moves them in the Y direction, but there is no reason you could not use a single FOR statement to shift a polygon in both the X and Y dimensions at the same time.

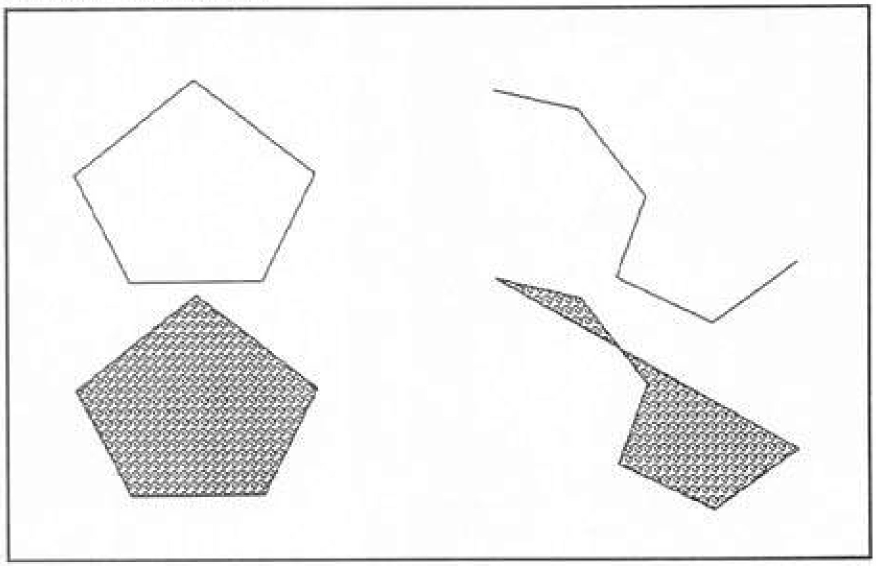
```
{--------------------------------------------------------------}
{                         Polygons                             }
{                                                              }
{       Polygon draw/polygon fill demonstration program        }
{                                                              }
{                             by Jeff Duntemann                }
{                             Turbo Pascal V5.0                }
{                             Last update 7/14/88              }
{                                                              }
{                                                              }
{                                                              }
{--------------------------------------------------------------}

PROGRAM Polygons;

USES Graph;

```

```
CONST
  Squiggles : FillPatternType =
              ($94,$84,$48,$30,$00,$c1,$22,$14);

  Pentagon  : ARRAY[0..5] OF PointType =
              ((X : 100; Y :  10),
               (X : 200; Y :  80),
               (X : 155; Y : 160),
               (X :  45; Y : 160),
               (X :   0; Y :  80),
               (X : 100; Y :  10));

  BigDipper : ARRAY[0..5] OF PointType =
              ((X : 350; Y :  20),
               (X : 420; Y :  35),
               (X : 475; Y : 100),
               (X : 450; Y : 160),
               (X : 530; Y : 195),
               (X : 600; Y : 150));


VAR
  I           : Integer;
  GraphDriver : Integer;
  GraphMode   : Integer;
  ErrorCode   : Integer;


BEGIN

  GraphDriver := Detect;  { Let the BGI determine what board we're using }
  InitGraph(GraphDriver,GraphMode,'');
  IF ErrorCode <> 0 THEN
    BEGIN
      Writeln('>>Halted on graphics error: ',GraphErrorMsg(ErrorCode));
      Halt(2)
    END;

  SetFillPattern(Squiggles,White);

  DrawPoly(6,Pentagon);
  FOR I := 0 TO 5 DO     { Translate the pentagon down 160 pixels }
    Pentagon[I].Y := Pentagon[I].Y + 160;
  FillPoly(6,Pentagon);

  DrawPoly(6,BigDipper);
  FOR I := 0 TO 5 DO     { Translate the Big Dipper down 140 pixels }
    BigDipper[I].Y := BigDipper[I].Y + 140;
  FillPoly(6,BigDipper);

  Readln;
  CloseGraph;
END.
```

Below is a screen shot of the patterns produced by POLYGON.PAS. Note the way that the Big Dipper figure was filled by the BGI, as two separate closed figures formed when the two endpoints of the figure were connected by the shortest line between them.

Figure 22.8

Open and Closed Polygons



Drawing Rectangles and Bars

We've already used the BGI's **Rectangle** procedure informally. It draws an outline rectangle in the current line style and color:

```
PROCEDURE Rectangle(X1,Y1,X2,Y2 : Integer);
```

where **X1,Y1** specify the upper left corner of the rectangle, and **X2,Y2** specify the lower right ~~left~~ corner of the rectangle. **Rectangle** is viewport relative.

Drawing a filled rectangle is done with the **Bar** procedure:

```
PROCEDURE Bar(X1,Y1,X2,Y2 : Integer);
```

Bar is basically identicial to **Rectangle**, except that the interior of the rectangle is filled with the current fill color or pattern, as set by **SetFillStyle** or **SetFillPattern**. The perimeter of the bar is *not* drawn; only the interior portion is generated in the fill color or pattern, and the pattern goes out to the edges. If you want a line perimeter around your bars, you must draw them separately, with the **Rectangle** statement, *after* the bar is drawn. If you drawn the perimeter first with **Rectangle** and *then* try to fill the rectangle with the **Bar** statement, you'll discover that your perimeter disappears under the fill color or pattern. Like **Rectangle**, **Bar** is viewport-relative.

Far more interesting than either **Rectangle** or **Bar** is **Bar3D**:

```
PROCEDURE Bar3D(X1,Y1,X2,Y2 : Integer;
                Depth       : Word;
                Top         : Boolean);
```

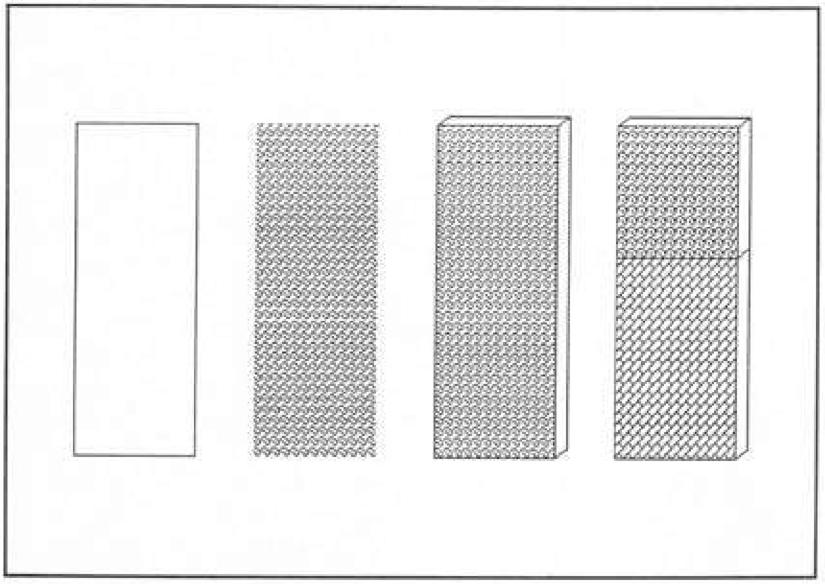
The idea is to facilitate the generation of 3-D bar graphs, which are very popular with the desktop-presentation set. The first four parameters operate identically to those of **Rectangle** and **Bar**, and set the upper left corner and lower right corner of the front-facing rectangle of the 3-D bar to be drawn. The **Depth** parameter specifies the depth of the 3-D bar in pixels. The **Top** parameter specifies whether the top portion of the bar's 3-D extension is to be drawn. This becomes important if you want to stack bars of different colors or fill patterns, as is often done in bar graphs to indicate different meanings for portions of the same numeric figure.

Unlike **Bar**, an outline *is* drawn around the filled portion of the bar. This outline is drawn in the current line color and style.

The best way to appreciate **Bar3D** is to see it operate. Figure 22.9 shows the BGI's rectangle family. Note the two stacked bars filled with different patterns. The bottom bar was drawn with **Top** set to **False**, so that the top of the bar was not drawn. The top bar was drawn with **Top** set to **True**, so that a top was drawn, properly finishing it off as the top of the entire bar.

Figure 22.9

Rectangular Figures



The four figures shown in Figure 22.9 were drawn on an EGA screen with the statement sequence below. The two patterns, **Squiggles** and **Bricks**, were defined in the **Patterns** program given in the previous section.

```
SetFillPattern(Squiggles,White); { Set a fill pattern }
Rectangle(50,50,150,300);        { Draw an outline rectangle }
Bar(200,50,300,300);             { Draw a filled bar }
Bar3D(350,50,450,300,10,True);   { Draw a filled 3-D bar }
Bar3D(500,50,600,148,10,True);   { Draw top portion of 2-part bar }
SetFillPattern(Bricks,White);    { Change fill pattern }
Bar3D(500,148,600,300,10,False); { Draw bottom portion of 2-part bar }
```

Drawing Ellipses and Elliptical Arcs

Drawing curved figures of any kind is always harder than creating things out of straight lines. The BGI does a tolerable job of certain kinds of curved figures, although it has no general procedure for drawing curved lines. Specifically, the BGI draws ellipses, circles (which are special cases of the more general ellipse), portions of ellipses or circles called arcs, and filled segments of circles called *pie slices* from their only real use in generating the ever-popular pie chart.

The most general routine for drawing curved figures with the BGI draws ellipses and elliptical arcs:

```
PROCEDURE Ellipse(X,Y : Integer; StartAngle,EndAngle : Word;
                  XRadius,YRadius : Word);
```

The **X** and **Y** parameters specify a "center" point for the ellipse or arc. If you know geometry this might seem odd, since ellipses actually have two centers. In the context of the BGI, however, the center of a drawn ellipse is the midpoint of the major axis. This major axis is always either horizontal or vertical; it is not possible to draw an ellipse on an oblique axis (see Figure 22.10).

Specifying the shape of an ellipse must be done by passing **Ellipse** two separate radius figures. **XRadius** specifies the distance in pixels from the center of the ellipse to the edge of the ellipse along the X (horizontal) axis. **YRadius** specifies the distance from the center of the ellipse to its edge along the Y (vertical) axis. Bullet 1 in Figure 22.10 shows a complete ellipse with its relation to the center point at X,Y and the two radius figures, **XRadius** and **YRadius**.

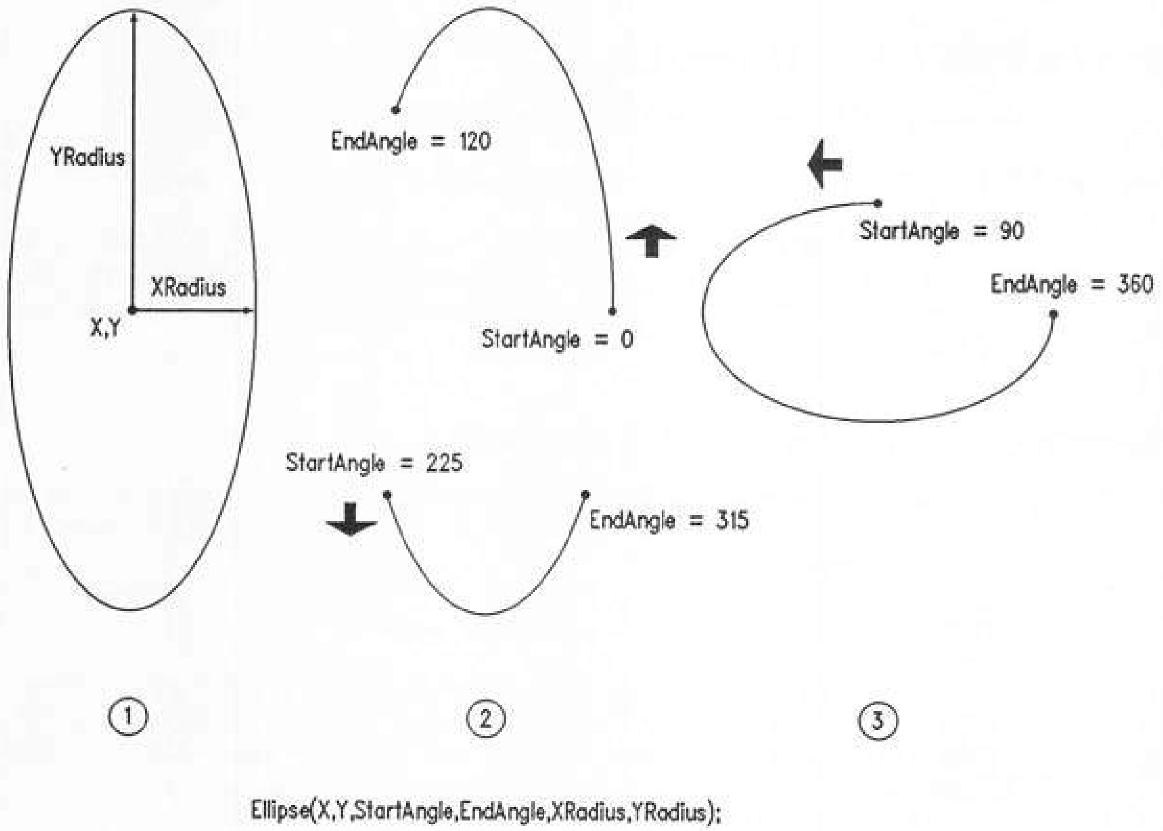
The ellipse will be drawn in the current drawing color; however, ellipses do not use styled lines and will be drawn only in a single-pixel solid line.

There is no generalized curve-drawing routine in the BGI for drawing spline curves. However, the **Ellipse** procedure is versatile enough to draw elliptical arcs, which, with some cleverness and patience, may be pieced together to form continuous curves.

An elliptical arc is nothing more than a portion of the perimeter of an ellipse. Bullet 2 in Figure 22.10 shows two such portions of the same ellipse shown in Bullet 1.

Figure 22.10

The Structure of Ellipses and Arcs



By varying the axes of the ellipse and the portion of the ellipse "lifted" out as a curve, most simple curved segments can be produced.

Specifying the portions of an ellipse drawn as elliptical arcs is done with angular measurement. The ellipse is divided into 360 degrees, with the 0 point at 3 o'clock. The degree measurement increases in a counterclockwise direction, all the way around the ellipse until the 360 degree point, which is the same as the 0 point. Thus, 90 degrees is at 12 o'clock, 180 degrees at 9 o'clock, 270 degrees at 6 o'clock, and so on. The upper curve at Bullet 2 in Figure 22.10 starts at 0 degrees and continues to 120 degrees. The curve may start anywhere along the perimeter of the ellipse; the bottom curve at Bullet 2 begins at 225 degrees and continues to 315 degrees.

As with complete ellipses, elliptical arcs are drawn only in a solid, single-pixel width line. Line styles and thicknesses do not operate at all within the **Ellipse** procedure.

Portions of a drawn ellipse extending beyond the bounds of the current clipping viewport or beyond the physical screen are clipped and not displayed.

Drawing Filled Ellipses (Version 5.0)

Creating filled ellipses is done with the following 5.0-specific procedure:

```
PROCEDURE FillEllipse(X,Y : Integer; XRadius,YRadius : Word);
```

Functionally, **FillEllipse** is identical to **Ellipse** minus **Ellipse**'s ability to draw elliptical arcs. (Filling an arc is meaningless, so the **StartAngle** and **EndAngle** parameters are not present.) The ellipse, when drawn, is filled with the current fill pattern using scan conversion.

Drawing Circles, Circular Arcs, and Pie Slices

In geometry, a circle is actually a special case of an ellipse. The same is true to some extent in the BGI. Circles are needed often enough to warrant their having a procedure all to themselves:

```
PROCEDURE Circle(X,Y : Integer; Radius : Word);
```

Circles are simpler creatures than ellipses. The **X,Y** parameters specify the coordinates of the circle's center point, and **Radius** specifies the radius of the circle in pixels. The circle is drawn using the current drawing color, but, as with **Ellipse**, line styles do not operate on circles.

The BGI supplies a separate routine for drawing circular arcs:

```
PROCEDURE Arc(X,Y : Integer; StartAngle,EndAngle,Radius : Word);
```

A circular arc is simply a segment of a circle. The **X,Y** parameters, again, specify the center point of the circle of which the arc is a segment. **Radius** is the radius of the arc. **StartAngle** and **EndAngle** operate precisely as they do in **Ellipse**: They indicate the angular portion of the circle which the arc represents. The 0 point is at 3 o'clock, and so on, just as for **Ellipse**.

Why are there separate routines for drawing circles and arcs? Over the years, computer scientists have developed tricky methods for drawing circles very quickly that depend on the circle's radial symmetry. Only *one eighth* of the circle is actually calculated; the remaining seven sections are plotted as "reflections" of the single calculated segment. In drawing an arc, the code does not know up front what angular portion of the arc will be drawn. It must be ready to calculate and draw all portions of the arc, and for a sizeable arc this could take much longer than for a circle of equal radius. In short, the BGI draws circles about as quickly as it is possible to draw them; arcs, being a messier problem to solve, are drawn with their own procedure.

As with circles and ellipses, circular arcs are drawn in the current drawing color but without adhering to the current line style.

The BGI allows programs to query the coordinates of the last arc drawn using the **Arc** procedure:

```
PROCEDURE GetArcCoords(VAR ArcCoords : ArcCoordsType);
```

The procedure returns a record type that is predefined within the **Graph** unit:

```
TYPE
  ArcCoordsType = RECORD
                    X,Y : Integer;
                    XStart,YStart : Word;
                    Xend,YEnd : Word
                  END;
```

The **X,Y** fields are the coordinates of the center point of the last arc drawn. **XStart,YStart** are the coordinates of the starting point of the arc, and **XEnd,YEnd** are the coordinates of the last point drawn. All coordinates are relative to the current viewport.

Why is this useful? Remember that when you specify an arc, you don't specify the coordinates for the starting and ending points. You specify *angles*. The BGI then calculates what pixels on the screen these angles represent. If you want to merge a straight line seamlessly with the end of an arc, you need to know the exact X,Y coordinates of the end in question.

A good example of how this feature might be used lies in the drawing of *rounded rectangles*. Most people feel that rectangles with rounded corners are slightly more aesthetically pleasing than rectangles that come to points at all four corners. Drawing four 90-degree arcs as the corners of the a rectangle and then connecting the endpoints of the arcs with straight lines will produce a rounded rectangle. The following procedure generalizes the idea:

```
{->>>>RoundedRectangle<<<<------------------------------------}
{                                                            }
{ Filename : ROUNDRCT.SRC -- Last Modified 7/23/88           }
{                                                            }
{ This routine draws a rectangle at X,Y; Width pixels wide and }
{ Height pixels high; with rounded corners of radius R.      }
{                                                            }
{ The Graph unit must be USED for this procedure to compile. }
{                                                            }
{                                                            }
{                                                            }
{------------------------------------------------------------}

PROCEDURE RoundedRectangle(X,Y,Width,Height,R : Word);
```

```

VAR
  ULData,LLData,LRData,URData : ArcCoordsType;

BEGIN
  { First we draw each corner arc and save its coordinates: }
  Arc(X+R,Y+R,90,180,R);
  GetArcCoords(ULData);
  Arc(X+R,Y+Height-R,180,270,R);
  GetArcCoords(LLData);
  Arc(X+Width-R,Y+Height-R,270,360,R);
  GetArcCoords(LRData);
  Arc(X+Width-R,Y+R,0,90,R);
  GetArcCoords(URData);
  { Next we draw the four connecting lines: }
  Line(ULData.XEnd,ULData.YEnd,LLData.XStart,LLData.YStart);
  Line(LLData.XEnd,LLData.YEnd,LRData.XStart,LRData.YStart);
  Line(LRData.XEnd,LRData.YEnd,URData.XStart,URData.YStart);
  Line(URData.XEnd,URData.YEnd,ULData.XStart,ULData.YStart);
END;
```

A separate record of type **AcrCoordsType** is defined for each of the four corners. After each corner is drawn, its coordinates are immediately queried and stored in the appropriate record. Only when all four corners have been drawn are the straight lines drawn to connect the rounded corners. Can you explain why?

It is possible to put nonsense values for R (e.g., a radius larger than the rectangle is wide), but, aside from bizarre graphics on the screen, no harm will come of it. Keep in mind that if you have a line style in force, the straight lines in the drawn figure will use the line style whereas the rounded corners will not.

The BGI includes a routine that will draw a filled circle, or a *filled arc* more commonly called a *pie slice:*

```
PROCEDURE PieSlice(X,Y : Integer; StartAngle,EndAngle,Radius :
Word);
```

The parameter list here is identical to that of the **Arc** procedure, both in declaration and operation. An arc is drawn just as it would be using the **Arc** procedure, using the current drawing color. Next, its endpoints are joined to the arc's center point at **X,Y** by two straight lines. These lines will be drawn in the current drawing color *and current line style.* Finally, the region enclosed by the arc and the two radius lines is filled using the BGI's scan converter and the current fill color or pattern.

This provides a means of drawing a filled circle, assuming it is filled by the current drawing color:

```
PieSlice(100,100,0,360,40);
```

This statement will draw a circle filled with the current drawing color. Notice that the start angle is 0 and the end angle is 360, which indicates a complete circle. The problem with drawing filled circles arises if they are to be filled with a pattern or some color *other* than the current drawing color. The radius lines running between the center point and the end points of the arc will still be drawn, even if the arc specified is a complete circle. What you have is a line from the center of the circle to the 3 o'clock position, as shown in the uppermost of the two filled circles in Figure 22.11.

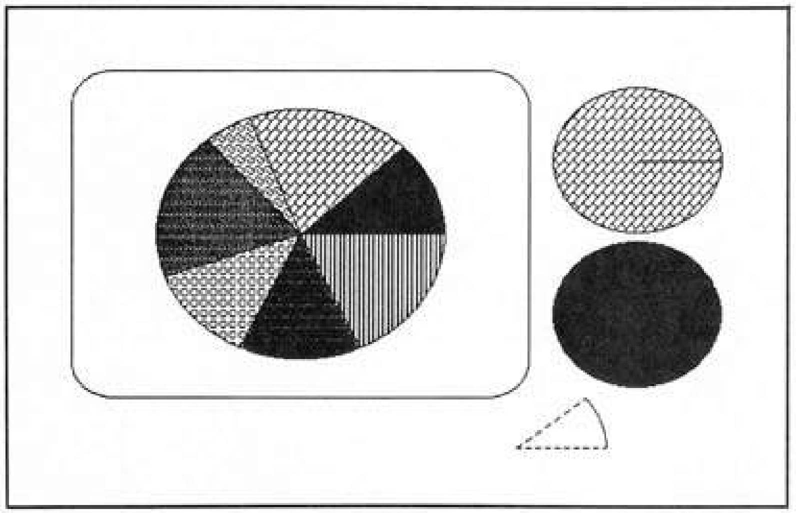
The way around this is to use the **SetColor** procedure to set the current drawing color to the same color as the fill color you specify with **SetFillStyle.** This will yield the bottom circle of the pair in Figure 22.11.

Oddly, the radius lines drawn as part of the pie slices will be drawn with the current line style, even though the curved portion of the arc around the edges of the slice does *not* use the current line style. Figure 22.11 shows an "empty" pie slice (i.e., one filled with the current background color) with its radius lines exhibiting the current line style.

The only important use of **PieSlice** lies in generating segments of filled circles for inclusion in pie chart graphs. Figure 22.11 shows such a chart, with seven separate pie slices sharing a common center point, and each filled with a different fill pattern. The box around the pie chart was drawn with the **RoundedRectangle** procedure presented earlier.

Figure 22.11

Pie Slices in Various Uses



The program below, PIEMAN.PAS, generated the graphics shown in Figure 22.11.

```
{--------------------------------------------------------------}
{                          PieMan                              }
{                                                              }
{               PieSlice demonstration program                 }
{                                                              }
{                              by Jeff Duntemann               }
{                              Turbo Pascal V5.0               }
{                              Last update 7/14/88             }
{                                                              }
{                                                              }
{                                                              }
{--------------------------------------------------------------}

PROGRAM PieMan;

USES Graph;

CONST       { Fill Patterns for pie chart: }
  Halftone1 : FillPatternType =
              ($CC,$33,$CC,$33,$CC,$33,$CC,$33);
  Halftone2 : FillPatternType =
              ($AA,$55,$AA,$55,$AA,$55,$AA,$55);
  Squiggles : FillPatternType =
              ($94,$84,$48,$30,$00,$c1,$22,$14);
  Vertical  : FillPatternType =
              ($CC,$CC,$CC,$CC,$CC,$CC,$CC,$CC);
  Bricks    : FillPatternType =
              ($01,$82,$44,$28,$10,$20,$40,$80);
  Blocks    : FillPatternType =
              ($00,$3C,$42,$42,$42,$42,$3C,$00);


VAR
  GraphDriver : Integer;
  GraphMode   : Integer;
  ErrorCode   : Integer;


{$I ROUNDRCT.SRC}  { RoundedRectangle }


BEGIN
  GraphDriver := Detect;  { Let the BGI determine what board we're using }
  InitGraph(GraphDriver,GraphMode,'');
  IF ErrorCode <> 0 THEN
    BEGIN
      Writeln('>>Halted on graphics error: ',GraphErrorMsg(ErrorCode));
      Halt(2)
    END;

  RoundedRectangle(30,30,380,260,35);  { Draw the pie graph frame }

  PieSlice(220,160,0,45,120);          { Draw the pie chart segments }
  SetFillPattern(Bricks,White);
  PieSlice(220,160,45,110,120);
  SetFillPattern(Squiggles,White);
```

```
    PieSlice(220,160,110,130,120);
    SetFillPattern(Halftone1,White);
    PieSlice(220,160,130,200,120);
    SetFillPattern(Blocks,White);
    PieSlice(220,160,200,245,120);
    SetFillPattern(Halftone2,White);
    PieSlice(220,160,245,295,120);
    SetFillPattern(Vertical,White);
    PieSlice(220,160,295,360,120);

    SetFillStyle(SolidFill,White); { Set White as fill color }
    PieSlice(500,220,0,360,70);    { Draw a color-filled circle }
    SetFillPattern(Bricks,White);  { Set a fill pattern }
    PieSlice(500,105,0,360,70);    { Draw a pattern-filled circle }

    SetLineStyle(3,0,1); SetFillStyle(EmptyFill,White);
    PieSlice(400,320,0,40,75);

    Readln;
    CloseGraph;
  END.
```

Drawing Elliptical Sectors (Version 5.0)

Turbo Pascal 5.0 extends the concept of a circular pie slice to an elliptical pie slice, more formally called a *sector*. A sector is a portion of an elliptical arc with the endpoints of the arc joined to the center of the ellipse. In essence, it is a pie slice with an X radius and a Y radius:

```
PROCEDURE Sector(X,Y : Integer;
                 StartAngle,EndAngle,
                 XRadius,YRadius : Word);
```

As with **PieSlice, Sector** fills the region enclosed by the arc with the current fill color and style.

22.6: BIT FONTS AND STROKE FONTS

The word *font* is a relatively recent addition to the IBM PC vocabulary, because until fairly recently there was only one font available: the one embedded in ROM on the display board. The EGA graphics board had the ability to load several text fonts, but no software really took advantage of the power. With the appearance of faster machines and more sophisticated graphics software, fonts have become an essential part of graphics programming. The BGI's font support is considerable, and in this section we'll examine it in detail.

Bit Fonts

A "bit font" is a font defined as a pattern of individual pixels, each one represented by one bit in a table somewhere in memory. All text-based fonts are bit fonts and are generally burned into Programmable Read Only Memory chips (PROMs) on the graphics board. The BGI provides a bit font for use in graphics modes. The character patterns in the BGI's bit font are limited to a "character cell" 8 pixels high and 8 pixels wide. The bit font is thus called an *8 × 8* font.

An 8 × 8 font may seem small, but it can be made larger by representing each pixel in the character patterns by four pixels arranged in squares. Therefore, enlarging it means doubling its size at each level of enlargement, which is not a great deal of control. Furthermore, as the font gets larger, it starts looking "blockier"; and it is, of course, harder to read and less attractive.

Bit fonts have the advantage that they can be displayed on the screen very rapidly, since displaying a bit font character is only a matter of moving eight bytes from one location in memory (the character pattern table) to another (the video display buffer).

Stroke Fonts

Another kind of font is the *stroke font*, so called because its characters are defined as sequences of straight lines or *strokes*, as of the stroke of a pen. Stroke fonts can be made very elaborate, almost calligraphic and ornamental, and very pleasing compared to the blockier bit fonts. As they are composed of straight lines, stroke fonts are easily scaled up or down in size in very fine increments, simply by making each line comprising each character a little bit longer or a little bit shorter, with everything in proportion. These stylistic and scaling features come at the cost of speed, since every character requires several (or perhaps 15 or 20) line-draw operations. Also, because the lines on a typical PC graphics device are relatively coarse, stroke font characters can only be made so small before they blur into unintelligible blobs. In use, they tend to be much larger on the screen than bit font characters, more suitable for titles and banners on presentation slides than for dense text.

The BGI's stroke fonts are derived from the Hershey fonts, a well known set of stroke font definitions developed by the Navy in the early days of computing. At this writing only four stroke fonts are available with the BGI, but others will be made available in time, and Borland plans to release the font file specification so that we can define our own.

Setting Font Type, Size, and Direction

A BGI font has three characteristics: font type (i.e., which font from the BGI's repertoire); font size on the screen; and the direction the text is to read from left to right in a horizontal direction, or from bottom to top in a vertical direction. Fonts are drawn in the current drawing color, but even though stroke fonts are drawn with straight lines, they are *not* affected by the current line style.

Setting font parameters is done with a single procedure:

```
PROCEDURE SetTextStyle(Font,Direction,CharSize : Word);
```

"Normal" size for a bit font is 1, and for a stroke font is 4. Stroke fonts will display with a **CharSize** value from 1 through 3, but they will rarely be readable and are almost never attractive. The largest value accepted for any font is 10. Passing a value larger than 10 in **CharSize** will not trigger any sort of error. However, the BGI will treat the oversize value as though it were the value 10, and the characters will be drawn no larger than they would if 10 were passed in **CharSize** instead.

For the other two parameters, a number of constants have been defined to make dealing with raw numeric codes a little easier. Table 22.8 summarizes the constants as defined in the BGI.

Table 22.8
BGI Font Parameter Constants

Constant	*Value*	*Parameter*	*Meaning*
DefaultFont	0	Font	The BGI's default 8 × bit font
TriplexFont	1	Font	"Three line" serif stroke font
SmallFont	2	Font	Highly reduceable stroke font
SansSerifFont	3	Font	"Two-line" font without serifs
GothicFont	4	Font	"Ye Olde English" ornamental font
HorizDir	0	Direction	Left to right, horizontally
VertDir	1	Direction	Bottom to top, vertically

The **Direction** parameter indicates which way the text will be displayed on the screen. With **Direction** set to 0, text will be displayed normally, from left to right in a horizontal direction. When **Direction** is set to 1, text will be displayed vertically, starting from the bottom and going toward the top of the screen.

The available BGI fonts are easier to display than describe. I've shown a sampling of all fonts in Figure 22.12. Font 2, **SmallFont**, is an attempt to define a stroke font that is readable and tolerably attractive in small sizes. Unlike most of the other stroke fonts, it loses something with a gain in size. It begins to look slightly bizarre when enlarged, because it is composed of relatively few strokes, and individual characters come to resemble something off the cover of a heavy metal rock album. It is the only stroke font that can hold its own at the same size as the default font 0. Keep in mind, though, that it takes longer to display text with **SmallFont** than with **DefaultFont**, even when they are scaled to approximated the same size.

GothicFont is highly ornamental and difficult to read at any size. Its deliberately antique nature clashes severely with the high-tech look of a graphics application on a color screen. The two best fonts for use at fairly large sizes are **TriplexFont** and **SansSerifFont**, which differ almost entirely in that one has serifs and the other doesn't. A "serif" in font jargon is one of those little swellings at the extremities of certain characters; for example, at the bottom of the uppercase T and lowercase p. Fonts without serifs (called "sans serif" fonts, "sans" being French for "without") look simpler and more modern and may be slightly easier to read than fonts with serifs. Apart from that, serifs matter only in questions of aesthetics.

Font Initialization Errors

When you execute **SetTextStyle**, the BGI must load and prepare a font file for use. This means that it is at the mercy of your disk system. If the font is not found on the disk, or if the drive is not ready for access, errors will be generated. You should check **GraphResult** after calling **SetTextStyle** to see if all went well. Possible errors are summarized in Table 22.9.

The default font (font 0) is the only font that does *not* need to be loaded from a disk file. Its character patterns are part of the BGI and are always kept in memory. You should keep in mind that alternating rapidly between stroke fonts will cause delays as the fonts are loaded and reloaded repeatedly from disk.

Figure 22.12

The BGI Standard Fonts

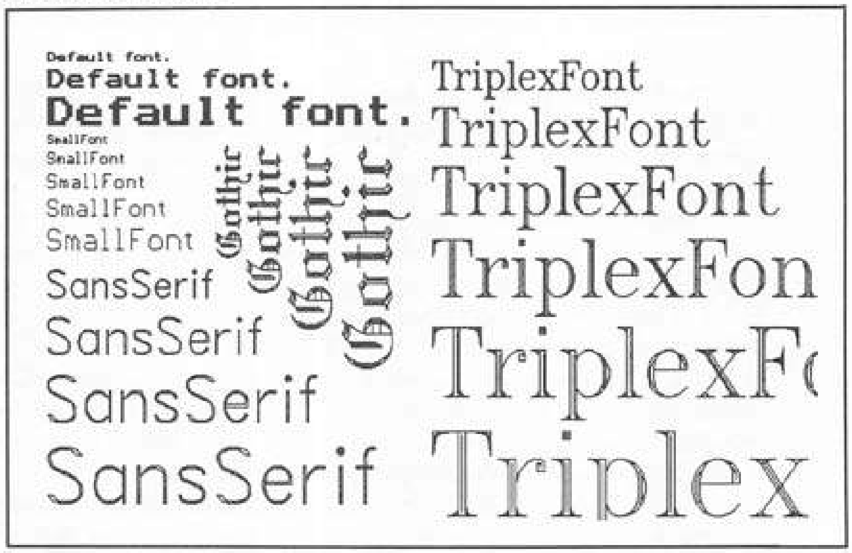


Table 22.9
Possible Font Initialization Errors

Error	*Meaning*
−8	Font file not found
−9	Not enough memory to load the font selected
−12	Graphics I/O error
−13	Invalid font file
−14	Invalid font number

Displaying Text on the Screen

Two routines are provided for actually displaying graphics text on your screen:

```
PROCEDURE OutText(TextString : String);

PROCEDURE OutTextXY(X,Y : Integer; TextString : String);
```

They differ only in how the position of the text on the screen is determined. **OutText** places the text at the current pointer (CP), while **OutTextXY** places the text at the coordinates passed in **X,Y**.

As you might have begun to think, in looking at Figure 22.12, there's a little more to it than that. A text string executed with a variable-size font occupies a varying amount of space on the screen. Where is the positioning point? And where does an **OutText** operation leave the CP?

You have to learn to think of displaying text under the BGI as a process of generating a rectangular box full of graphics and then positioning it somewhere on the screen. I call this the *text box*, even though the "box" is invisible and only the text is ever actually displayed. As you might expect, the positioning point when text is displayed through **OutTextXY** is the upper left corner of the rectangular box occupied by the graphics text. **OutTextXY** does *not* update the CP.

The interaction of CP with **OutText** and the text box is a little more complex. CP is the positioning point, but the relationship of the text box to CP depends on two parameters set with yet another BGI routine:

```
PROCEDURE SetTextJustify(H,V : Word);
```

Both **H** and **V** accepts values ranging from 0 through 2. As with most numeric codes used in the BGI, the **Graph** unit includes several named constants to make them easier to remember (Table 22.10).

Figure 22.13

The Relation of CP to the Text Box

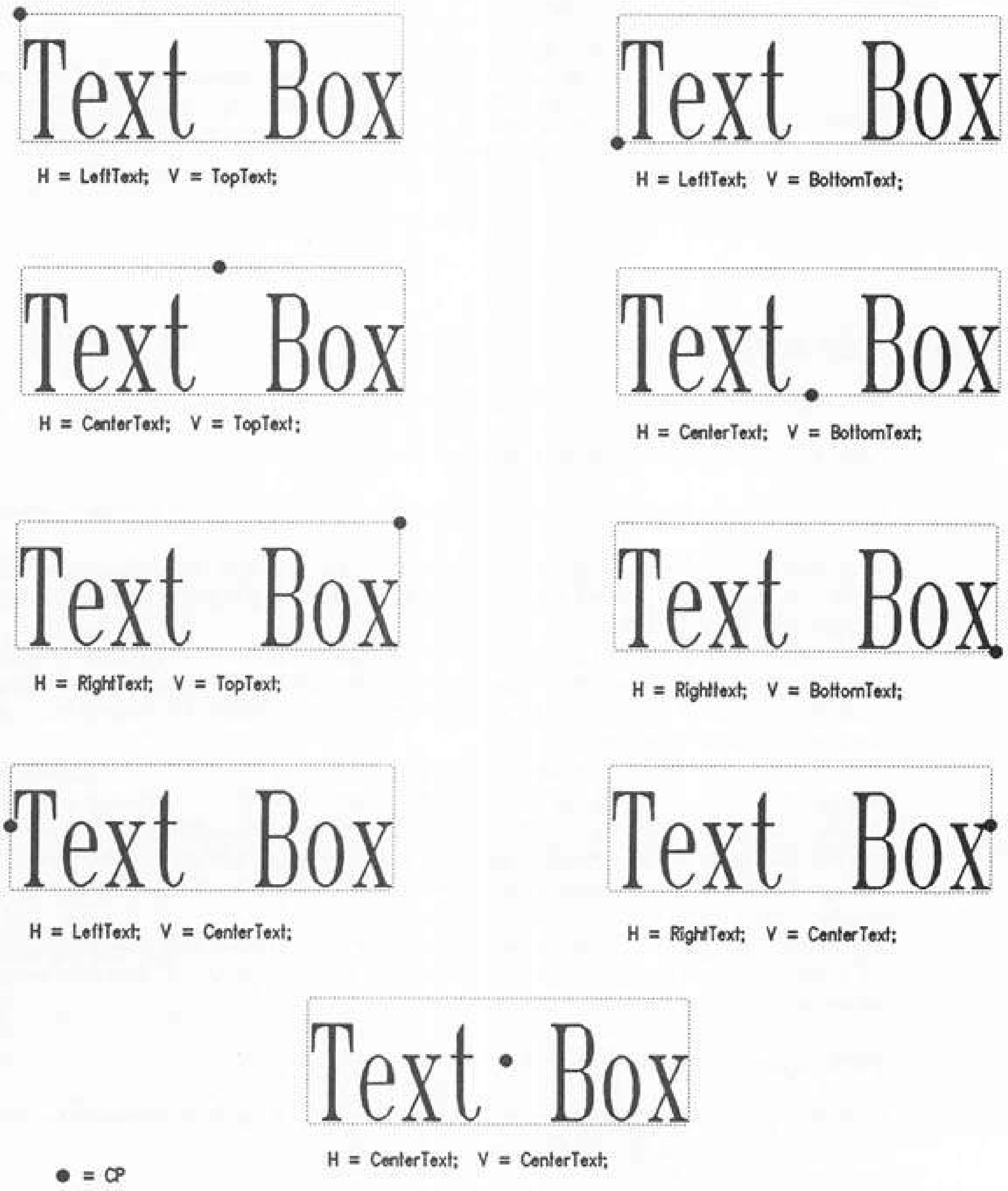

Text Box
H = LeftText; V = TopText;
Text Box
H = LeftText; V = BottomText;
Text Box
H = CenterText; V = TopText;
Text Box
H = CenterText; V = BottomText;
Text Box
H = RightText; V = TopText;
Text Box
H = Righttext; V = BottomText;
Text Box
H = LeftText; V = CenterText;
Text Box
H = RightText; V = CenterText;
Text Box
H = CenterText; V = CenterText;
● = CP

Table 22.10
Text Justification Constants

Constant	*Value*	*Parameter*	*Meaning*
LeftText	0	H	CP at left edge of text box
CenterText	1	H or V	CP centered in text box
RightText	2	H	CP at right edge of text box
BottomText	0	V	CP at bottom of text box
TopText	2	V	CP at top of text box

H defaults to 0 and **V** defaults to 1. Therefore, unless you change the defaults with a call to **SetTextJustify**, **OutText** positions the text box at the upper left corner of CP.

As with many matters of computer graphics, the situation is easier to show than to explain. Figure 22.13 shows the relationship of the CP and the text box for various combination of **H** and **V**. The question that follows is where CP goes after a call to **OutText**. Again, it depends on the combination of factors set by **SetTextStyle** and **SetTextJustify**.

First of all, when you set **Direction** to **VertDir** and write text in a vertical direction, the CP is *never* altered by **OutText**. When writing in a horizontal direction (**Direction** set to **HorizDir**) the CP is moved *only* when **H** is set to **LeftText**. **V** may be set to any of its legal values, but as long as **H** is set to **LeftText**, the CP will be moved to the *right* edge of the text box at whatever orientation specified by **V**. In other words, if **V** is set to **CenterText**, then CP will be moved from the left edge of the text box to the center of the right edge of the text box.

The program FONTS.PAS is given below as an example of how text is written to the screen in various fonts and various sizes, using the **OutTextXY** procedure.

```
{--------------------------------------------------------------}
{                           Fonts                              }
{                                                              }
{          BGI graphics font demonstration program             }
{                                                              }
{                             by Jeff Duntemann                }
{                             Turbo Pascal V5.0                }
{                             Last update 7/24/88              }
{                                                              }
{                                                              }
{                                                              }
{--------------------------------------------------------------}

PROGRAM Fonts;

USES Graph;

VAR
```

```
  I,J,K        : Integer;
  GraphDriver : Integer;
  GraphMode   : Integer;
  ErrorCode   : Integer;
  TestString  : String;


BEGIN

  GraphDriver := Detect;  { Let the BGI determine what board we're using }
  InitGraph(GraphDriver,GraphMode,'');
  IF ErrorCode > 100 THEN
    BEGIN
      Writeln('>>Halted on graphics error: ',GraphErrorMsg(ErrorCode));
      Halt(2)
    END;

  { Demonstrate the default font: }
  J := 0;
  FOR I := 1 TO 3 DO
    BEGIN
      SetTextStyle(DefaultFont,HorizDir,I);
      OutTextXY(0,J,'Default font.');
      J := J + 5 + TextHeight('DefaultFont');
    END;

  { Demonstrate the small font: }
  J := 60; TestString := 'SmallFont';
  FOR I := 4 TO 8 DO
    BEGIN
      SetTextStyle(SmallFont,HorizDir,I);
      OutTextXY(0,J,TestString);
      J := J + 5 + TextHeight(TestString);
    END;

  { Demonstrate the Triplex font: }
  J := 0; TestString := 'TriplexFont';
  FOR I := 4 TO 10 DO
    BEGIN
      SetTextStyle(TriplexFont,HorizDir,I);
      OutTextXY(320,J,TestString);
      J := J + 5 + TextHeight(TestString);
    END;

  { Demonstrate the SansSerif font: }
  J := 155; TestString := 'SansSerif';
  FOR I := 4 TO 10 DO
    BEGIN
      SetTextStyle(SansSerifFont,HorizDir,I);
      OutTextXY(0,J,TestString);
      J := J + 5 + TextHeight(TestString);
    END;

  { Now give poor weird Gothic a chance: }
  J := 120; TestString := 'Gothic';
  FOR I := 4 TO 7 DO
    BEGIN
      SetTextStyle(GothicFont,VertDir,I);
      OutTextXY(J,70,TestString);
      J := J + 5 + TextHeight(TestString)
    END;

  Readln;
  CloseGraph;
END.
```

The Size of the Text Box

Sometimes, deciding where to position text on the screen depends heavily on how large the text box is. The BGI provides two routines for returning the size of the text box, given a string of text:

```
FUNCTION TextWidth(TextString : String) : Word;

FUNCTION TextHeight(TextString : String) : Word;
```

Both routines take the text style into account when making their calculations. In other words, the values you pass to the BGI through the procedure **SetTextStyle** are used to calculate the dimensions of the text box.

The value returned by **TextWidth** is always the X-dimension of the text box, even when the text is being displayed in a vertical direction. Similarly, the value returned by **TextHeight** is always the Y-dimension of the text box.

One other procedure can affect the size of the text box by altering the vertical-to-horizontal proportions of stroke characters:

```
PROCEDURE SetUserCharSize(MultX,DivX,MultY,DivY : Word);
```

The idea here is to compress or expand stroke font characters in one dimension without affecting the other dimension. You can make stroke characters shorter and fatter, or taller and thinner, by passing the correct values in the four parameters.

The parameters **MultX** and **DivX** act as a pair to specify a multiplier for the X-dimension, so as to change the width of stroke characters without changing their height. Think of the calculation this way:

```
                            MultX
New Character Width =  -------------  * Standard Character Width
                            DivX
```

Similarly, the parameters **MultY** and **DivY** act together to specify a multiplier for the Y dimension. They operate this way:

```
                             MultY
New Character Height = -------------  * Standard Character Height
                             DivY
```

For example, if you want to make the text twice as wide without altering the height, specify a value of 2 for **MultX** and 1 for **DivX**. Or, to make the text half as high without affecting its width, specify a value of 1 for **MultY** and 2 for **DivY**.

The default for all four parameters is 1. This is the normal proportion of width to height set for the stroke fonts. To return to the default, simply make this call:

```
SetUserCharSize(1,1,1,1);
```

Under Turbo Pascal 4.0, you must call **SetTextStyle** immediately after a call to **SetUserCharSize** to make the new character size multipliers take effect. Under 5.0 this is no longer the case: A call to **SetUserCharSize** takes effect immediately.

Clipping Cautions

Something you must keep in mind: Stroke fonts are clipped at the screen boundaries and at viewport boundaries. *Bit fonts are never clipped.* If you attempt to write a string to the display using a bit font and part of the string extends beyond the boundaries of the screen or current viewport, *none* of that string will be displayed. The **OutText** or **OutTextXY** procedure will have simply no effect.

This is an excellent reason to use the **TextHeight** and **TextWidth** functions to keep an eye on the size of your text when using a bit font. One little corner over the edge, and nothing will be displayed at all.

Querying BGI Text Settings

Your programs can query the BGI for the current state of the various text settings by calling this procedure:

```
PROCEDURE GetTextSettings(VAR TextInfo : TextSettingsType);
```

The type returned in the **VAR** parameter **TextInfo** is a record defined within the **Graph** unit:

```
TYPE
  TextSettingsType =
    RECORD
      Font      : Word;  { Font code 0-4 }
      Direction : Word;  { 0 = horizontal; 1 = vertical }
      CharSize  : Word;  { 1-10 }
      Horiz     : Word;  { 0=left; 1=center; 2=right }
      Vert      : Word   { 0 = bottom; 1 = top }
    END;
```

The fields contain either the default values or the values that were set with **SetTextStyle** and **SetTextJustify**.

Generating BGI Error Message Strings

The BGI very conveniently includes a function that returns a descriptive string for each error message:

```
FUNCTION GraphErrorMsg(ErrorCode : Integer) : String;
```

This makes it unnecessary to hard-code such strings into your own programs, assuming you consider the BGI's error messages adequate. Essentially, after a graphics operation that affects **GraphResult**, you can pass the value returned by **GraphResult** to **GraphErrorMsg** and then display the string in an error message window in the size, font, and screen location of your choosing. This is certainly better than having the BGI simply burst in with an error message of its own anywhere it chooses.

Using **GraphErrorMsg** is simple:

```
SetTextStyle(TriplexFont,HorizDir,6);
OpResult := GraphResult;
IF OpResult <> 0 THEN
  OutTextXY(MsgX,MsgY,'>>Error! '+ GraphErrorMsg(OpResult));
```

22.7: BITMAPS AND BITBLTS

So far in our discussion of the BGI, everything has been concerned with putting graphics information up on the screen. Nothing interacts with graphics information *after* it appears on the display with the exception of two procedures that manipulate *bitmaps.*

All the graphics devices supported by the BGI contain their own video memory, and they encode pixel information in that memory by relating bits in memory to pixels on the screen. We say that bit information in video memory somehow *maps* to pixels displayed on the screen. In most cases, a 1 bit indicates foreground color on the screen, whereas a 0 bit indicates background color. The EGA and VGA consist of several separate but congruent *bit planes,* each representing one of the primary colors. A 1 bit in the red bit plane means that the pixel corresponding to that position in the bit plane will contain red information. If the corresponding bit in the green bit plane also contains a 1, the pixel will contain both red and green information, and the color of the pixel will be brown.

You needn't understand this to make use of bitmaps, but some understanding of the background technology often helps. In short, a bitmap is a collection of bits in memory that represent some rectangular pattern of graphics on the screen. Functionally, you can think of a bitmap as a rectangular subset of the screen.

The BGI gives you the ability to "lift" a rectangular section of the graphics screen (we'll call those sections *bitmaps* from now on) from the display and store it in a variable, then "drop" the same bitmap back onto the display, perhaps at a different position. Lifting the bitmap from the display is done with this procedure:

```
PROCEDURE GetImage(X1,Y1,X2,Y2 : Integer; VAR BitMap);
```

Here, we pass the procedure two coordinate pairs that define a rectangular region of the screen, just as we pass to **Rectangle** or **Bar**. The **BitMap** parameter is untyped, and is therefore nothing but an address of a location in memory. When called, **GetImage** will read the bit information representing the rectangular region defined by the coordinates, and then store that information in memory starting at the address passed in **BitMap**.

This means that the actual parameter passed in **BitMap** had better be big enough to contain the graphics information, because no checks will be done. The BGI starts laying down bit information at the first byte of **BitMap** and continues until it's done. If your actual parameter is smaller than the information from the screen, adjacent storage may be overwritten.

The organization of the bitmap in memory is simple: The first word of the bitmap gives the width of the rectangular region stored as the bitmap, in pixels, counting from 1 and not 0. The second word is the height of the region in pixels, again counting from 1 and not 0.

The easiest way to lift a bitmap safely is to allocate storage for the bitmap on the heap. The **GetMem** procedure (see Section 10.7) allows you to reserve a region of the heap memory of a specified size and return a pointer to it. All that remains is finding out how much memory a given section of screen will occupy. The BGI has a function that will tell you:

```
FUNCTION ImageSize(X1,Y1,X2,Y2) : Word;
```

If you pass the coordinates that define a rectangular region of the screen to **ImageSize**, the function will return the number of bytes of storage that region will occupy when "lifted" by **GetImage**. To put it all together: storing a region of the screen on the heap is done this way:

```
VAR
  ImageChunk : Pointer;


GetMem(ImageChunk,ImageSize(0,0,100,50));
GetImage(0,0,100,50,ImageChunk^);
```

Laying down the bitmap elsewhere on the screen is done with another BGI routine:

```
PROCEDURE PutImage(X,Y : Integer; VAR BitMap; BitBlt : Word);
```

Here, the **X,Y** coordinates indicate the upper left corner of where the bitmap is to be positioned on the screen. Again, **BitMap** is the untyped parameter containing the stored bitmap. Now there is something new under the BGI sun: **BitBlt** (pronounced "bit blit"), a code specifying how the stored bitmap is to be logically combined with the information over which it will be written.

The word "bitblt" comes from the mnemonic for a machine-code instruction in an old DEC minicomputer meaning "bit block transfer." Understanding the notion of bitblt will be easier if you understand bit manipulation in general. I cover bitwise logical operations in general terms in Section 11.4 and in more detail in Section 23.5. You might read and digest those sections before taking on the **Bitblt** parameter of **PutImage**.

BitBlt's legal values and their meanings are summarized in the Table 22.11 below, along with the predefined constants from the **Graph** unit.

You might think of **PutImage** as a *two-dimensional* bitwise logical operator. You have a two-dimensional plane of bits stored on the heap (it was put there by **GetImage**) and another two-dimensional plane of bits in screen memory. **PutImage** combines the two, in accordance with the logical operator passed in the **BitBlt** parameter:

`NormalPut` (renamed **CopyPut** for Turbo Pascal 5.0) is a simple "bit move" operator. Each bit from the source plane replaces its corresponding bit in the screen plane. This replicates both foreground and background information on the screen. In essence, you have dropped the "lifted" bitmap whole onto its new position in the screen, completely replacing whatever was there before. This is probably the commonest mode for **PutImage**.

`XORPut` uses the logical exclusive OR operator. Each bit from the source plane is XORed with its corresponding bit in the screen plane. XOR is a slightly magical creature in its effects. If you XOR a bit plane over itself on the screen, all bits in the bitmap will be set to zero. In effect, you can erase a bitmap by first lifting it from the screen with **GetImage** and then using **PutImage** with **XORPut**

Table 22.11
Bitblt's Legal Values and Their Meanings

Constant	*Value*	*Meaning*
NormalPut	0	BitMap bits replace screen bits (V4.0)
CopyPut	0	Identical to NormalPut (V5.0)
XORPut	1	BitMap bits XOR with screen bits
ORPut	2	BitMap bits OR with screen bits
ANDPut	3	BitMap bits AND with screen bits
NOTPut	4	Negated BitMap bits replace screen bits

to lay it back down over itself. This is useful in animation. You can move a graphics object around the screen by XORing it at one position, XORing it over itself to erase it, and then XORing it back down again at a new position.

`ORPut` uses the logical OR operator. Each bit in the source plane is ORed with its corresponding bit in the screen plane. In effect, foreground information is transferred from **BitMap** to the screen, *but background information is not.* This is the way you would combine two images that overlap into a single larger image.

`ANDPut` uses the logical AND operator. Each bit in the source plane is ANDed with its corresponding bit in the screen plane. What happens here is that foreground bits will be transferred from **BitMap** to the screen *only* if their corresponding screen bits are also foreground bits. There isn't much use for this sort of bit combination, but it's available if you ever need it.

`NOTPut` uses the NOT operator, which inverts the bits in **BitMap** before moving them to the screen in the same way **NormalPut** does. In other words, each 1 bit from the source plane is inverted to a 0 bit and then replaces its corresponding bit in the screen plane; and each 0 bit from the source plane is inverted to a 1 bit, and then replaces its corresponding bit in the screen plane. Again, it is not tremendously useful, but **NOTPut** may be used to "flash" a window by lifting the window into memory with **GetImage**, dropping it back over itself with **PutImage** and **NOTPut**, and then putting it back down again with **PutImage** and **NormalPut**.

One very important use of **GetImage** and **PutImage** lies in menuing and windowing systems. "Popping up" a menu over a graphics background demands that the background be saved first. This is easily done by using **GetImage** to "lift" the background area onto the heap, then drawing the menu where the saved area had been. To restore the background area overwritten by the menu, you use **PutImage** to "drop" the saved area back down over the menu, covering it completely and making the screen look as it did before the menu "appeared."

I use this technique a great deal in the **PullDown** unit described in section 22.9. For examples of **GetImage** and **PutImage** in use, I refer you to that section.

Clipping Cautions

Clipping works a little oddly with respect to bitmaps. As with bit font output, if the entire bitmap cannot be laid down on the visible screen, it will not be laid down at all. There is one exception: the lower edge of the visible screen will clip a bitmap being laid down by **PutImage**. Viewports do not clip bitmaps at all.

22.8: GRAPHICS PAGING

A few of the BGI's supported graphics devices contain enough video memory to allow more than one distinct graphics screen. With more than one screen, "building" a complicated image can be done invisibly by directing graphics output to the screen that is not currently displayed. Then, at the appropriate time, the invisible screen can be made the visible screen, and the image will appear all at once.

One problem with this feature is that not all graphics devices support it, although many do. Currently, the only BGI-supported devices that include multiple graphics pages are the EGA, VGA, and Hercules. This means that to use paging in applications that may run on many different graphics devices, you must test for the presence of a multipage device before attempting to use paging.

Two procedures control the BGI's graphics paging feature:

```
PROCEDURE SetActivePage(Page : Word);

PROCEDURE SetVisualPage(Page : Word);
```

SetActivePage controls which graphics page the graphics output is written to. **SetVisualPage** controls which graphics page is currently visible:

```
SetVisualPage(1);          { Separate the visible from active pages }
SetActivePage(0);
DrawComplicatedThingie;  { Draw a complex image in the invisible page }
SetVisualPage(0);          { Now make it appear like magic! }
```

Values passed in **Page** must be legal for the current graphics device and mode. Table 22.12 sums up the various multipage devices and modes currently supported by the BGI, in terms of the values and constants used by **InitGraph** to select those modes.

Table 22.12
Multipage Devices and Modes

Constant	*Value*	*Meaning*	
EGALo	= 0	640 × 200; 2 color	4 pages
EGAHi	= 1	640 × 350; 16 color	2 pages
EGAMonoHi	= 3	640 × 350; 2 clr; 64K: 1 page;	256K: 4 pages
HercMonoHi	= 0	720 × 348; 2 color	2 pages
VGALo	= 0	640 × 200; 16 color	4 pages
VGAMed	= 1	640 × 350; 16 color	2 pages

If you pass an invalid page number to either of the paging control procedures, the operation will be ignored.

22.9: A GRAPHICS PULL-DOWN MENUING SYSTEM

There are two ways to design graphics support into an application. The low road is to write the application in text mode and enter graphics mode *only* when graphics-oriented work needs to be done. This is the design strategy of Borland's *Quattro* spreadsheet. The spreadsheet work is done entirely in text mode, with the business graphics feature accessible at the touch of a function key. The high road is simply to design the whole application in graphics, whether the immediate work being done is graphics-oriented or not. Graphics programs like Z-Soft's PC *Paintbrush* as well as nongraphics programs like Borland's *Reflex* go this route.

I'm of two minds about the issue. All-graphics applications are definitely the wave of the future, with the emphasis on *future*. In many respects, today's mainstream PC is not machine enough to handle the job acceptably. It isn't only a question of CPU speed. I have found that a 10-Mhz 8088 machine is not the equal of even an 8 Mhz 80286 machine in handling graphics. The bottleneck is in the display memory bandwidth, not CPU speed. The essence of doing graphics quickly is the moving of massive amounts of data around the video buffer in a hurry. CPU speed helps, but having to move only 1 memory byte per memory access (as the 8088 does) puts a choke hold on graphics performance. The 80286, being a true 16-bit machine, moves 2 bytes per memory access, and loses less CPU speed to bandwidth limitations. The 80386 and 80486 move data 4 bytes at a time, and may be the first Intel-family processors capable of doing justice to all-graphics applications. It's like the difference between emptying a bathtub with a teaspoon, a measuring cup, and a gallon bucket.

If you feel that you can afford to have your applications run haltingly on a 4.77-Mhz 8088 PC, by all means try the all-graphics route. The BGI is a great way to go about it, since all the really hard work (like designing and implementing drivers for the multitude of incompatible graphics devices out there) is done for you.

Mice, Menus, and Bouncing Bars

Over the past few years, machines like Apple's Macintosh have cemented certain ideas in the public mind about what an all-graphics application should look like. Most familiar is the notion of a pull-down menu user interface. A ubiquitous white bar across the top of the screen contains several key words, each of which represents a menu. These menus can be *pulled down* from the bar by depressing a mouse button while the mouse pointer points at or near the desired menu key word. When a menu is pulled down, a white *bounce bar* can be moved up or down a list of commands or other options by moving the mouse, highlighting whatever the bar covers in white. When the bar covers the desired option, the mouse button is released, and the option is considered selected.

If a menu is pulled down by mistake, or if the menu does not contain what the user wants to select, the mouse pointer is simply moved away from the menu, which then vanishes without further effect. It's an easily explorable, nonthreatening system of program control. An example of a pull-down menu system in use is shown in Figure 22.14.

The Menu as Data Structure

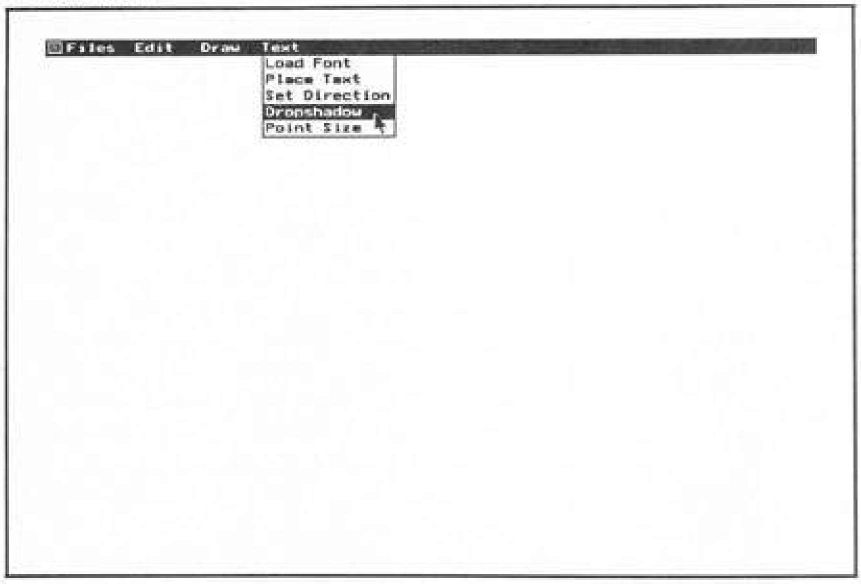
Designing such a menu system takes a certain amount of thought. It should be general enough so that it can be table driven. In other words, the names of the menus, their positions on the bar, the number of items in each menu, and the identifying code returned for each item should be part of some sort of data structure rather than hard-coded into the menuing routines themselves.

Such a data structure can be encapsulated in the data types shown below:

```
String15  = String[15];    { To keep the size of things down! }

ItemRec   = RECORD
              Item       : String15;
              ItemCode   : Byte;
              ItemActive : Boolean
            END;
```

Figure 22.14

A Pulldown Menu



```
MenuRec    = RECORD
               XStart,XEnd : Word;
               Title       : String15;
               MenuSize    : Word;
               Imageptr    : Pointer;
               Active      : Boolean;
               Choices     : Byte;
               ItemList    : ARRAY[0..18] OF ItemRec
             END;

MenuDesc   = ARRAY[0..12] OF MenuRec;
```

Within this system, an *item* is one of the choices in a *menu*, and a *menu* is a collection of items represented by an outlined box beneath the menu bar. Items are defined as variables of type **ItemRec**. Each item has a string that describes it on the screen (**Item**) and a numeric code that uniquely defines it (**ItemCode**). No other item within a given menu system may have that numeric code. When a given item is selected by the user, the menuing procedure returns that item's code to the calling logic.

The last item in the **ItemRec** definition is a Boolean flag that indicates whether or not the item is *active*; that is, whether the item will appear on the screen and be selectable at any given time. For example, in a text editing program, you might wish to disable any menu items involving editing of text until such time as the user selects and opens a file to edit. After the file is opened, editing again makes sense and the items should become available for selection. The menuing unit **PullDown** that I describe on page 531 includes procedures to activate and deactivate items at random at any time.

Items grouped together into a menu are part of a larger data structure embodying one menu of the several in the system as a whole. This structure is a record type called **MenuRec**. As with each item, the menu as a whole has a title, given by the field **Title**. This word or phrase is what appears in the white menu bar at the top of the screen. The user points to the title and depresses the left mouse button to pull down that menu. There is a region of the bar within which any left button click will pull down a menu. The left and right bounds of this region are given by the fields **XStart** and **XEnd**. The **Title** string is displayed within these bounds.

Like items, menus may be active or inactive during the execution of a program. The Boolean flag **Active** specifies whether the menu will be available for pulling down. The field **Choices** contains the number of items in the menu. An array called **ItemList** contains the items themselves, expressed as **ItemRec** records. Each menu may contain up to 19 items. The other fields in **MenuRec** we'll return to a little later, as they involve practical considerations of displaying the menu boxes quickly.

Finally, the collection of menus that are available across the menu bar is described by an array of **MenuRec** called **MenuDesc**. Up to 13 menus are possible in this system as I've implemented it. The maximum menu count across the bar (13) and the maximum number of items within a single menu (19) were compromises chosen to keep the total maximum number of items in the system under 255.

All the procedures that display and manipulate the menus are passed a variable of type **MenuDesc**. Now, **MenuDesc** is monstrous as data structures go (4,810 bytes in

832 fields) and if you define an ordinary variable of type **MenuDesc** you will somehow have to fill all 832 fields with their appropriate data. The practical alternative is to define **MenuDesc** structures as typed constants (see Section 9.6), which puts the burden of filling the structure with its data on the compiler, where it belongs. I have defined a single **MenuDesc** structure called **DemoMenu**, which is included on the listings diskette. It can be used as a boilerplate menu structure; modify it to create your own menus rather than typing in the entire monstrosity from scratch.

The alternative to using typed constants is to write the **MenuDesc** to disk as a binary file, and then read it back from disk into ordinary variables of type **MenuDesc**. This is less convenient than using typed constants, since it demands an editor of some sort that edits the binary data.

```
{->>>>DemoMenu<<<<------------------------------------------}
{                                                           }
{ Filename : DEMOMENU.DEF -- Last Modified 7/14/88          }
{                                                           }
{ This may be the largest structured constant you'll ever see. }
{ It defines a menu used by the PULLDOWN.PAS menu unit, which  }
{ implements a pulldown menuing system using the BGI.  You can }
{ copy this file to a new file and modify it to suit your      }
{ own applications.                                            }
{                                                           }
{ The MenuDesc type is defined in unit PullDown.            }
{                                                           }
{                                                           }
{                                                           }
{-----------------------------------------------------------}

CONST
  DemoMenu : MenuDesc =
             ((XStart    : 18; XEnd : 58;
               Title     : 'Files';
               MenuSize  : 11;
               Imageptr  : NIL;
               Active    : True;
               Choices   : 5; ItemList  :
               ((Item : 'Retrieve'; ItemCode : 21; ItemActive : True),
                (Item : 'Save';     ItemCode : 22; ItemActive : True),
                (Item : 'Delete';   ItemCode : 23; ItemActive : True),
                (Item : 'Rename';   ItemCode : 24; ItemActive : True),
                (Item : 'Quit';     ItemCode : 25; ItemActive : True),
                (Item : ''; ItemCode : 0; ItemActive : False),
                (Item : ''; ItemCode : 0; ItemActive : False),
                (Item : ''; ItemCode : 0; ItemActive : False),
                (Item : ''; ItemCode : 0; ItemActive : False),
                (Item : ''; ItemCode : 0; ItemActive : False),
                (Item : ''; ItemCode : 0; ItemActive : False),
                (Item : ''; ItemCode : 0; ItemActive : False),
                (Item : ''; ItemCode : 0; ItemActive : False),
                (Item : ''; ItemCode : 0; ItemActive : False),
                (Item : ''; ItemCode : 0; ItemActive : False),
                (Item : ''; ItemCode : 0; ItemActive : False),
                (Item : ''; ItemCode : 0; ItemActive : False),
                (Item : ''; ItemCode : 0; ItemActive : False),
                (Item : ''; ItemCode : 0; ItemActive : False))),
(XStart    : 74; XEnd       : 114;
 Title     : 'Edit';
 MenuSize  : 11;
 Imageptr  : NIL;
 Active    : True;
 Choices   : 9; ItemList  :
 ((Item : 'Grab';    ItemCode : 31; ItemActive : True),
  (Item : 'Pull';    ItemCode : 32; ItemActive : True),
  (Item : 'Erase';   ItemCode : 33; ItemActive : False),
  (Item : 'Join';    ItemCode : 34; ItemActive : False),
  (Item : 'Swap';    ItemCode : 35; ItemActive : False),
  (Item : 'Invert';  ItemCode : 36; ItemActive : True),
  (Item : 'Recolor'; ItemCode : 37; ItemActive : True),
  (Item : 'Split';   ItemCode : 38; ItemActive : True),
  (Item : 'Duplicate'; ItemCode : 39; ItemActive : True),
  (Item : ''; ItemCode : 0; ItemActive : False),
  (Item : ''; ItemCode : 0; ItemActive : False),
  (Item : ''; ItemCode : 0; ItemActive : False),
  (Item : ''; ItemCode : 0; ItemActive : False),
  (Item : ''; ItemCode : 0; ItemActive : False),
  (Item : ''; ItemCode : 0; ItemActive : False),
  (Item : ''; ItemCode : 0; ItemActive : False),
  (Item : ''; ItemCode : 0; ItemActive : False),
  (Item : ''; ItemCode : 0; ItemActive : False),
  (Item : ''; ItemCode : 0; ItemActive : False))),
(XStart    : 129; XEnd      : 175;
 Title     : 'Draw';
 MenuSize  : 11;
 Imageptr  : NIL;
 Active    : True;
 Choices   : 4; ItemList  :
 ((Item : 'Freehand';  ItemCode : 41; ItemActive : True),
  (Item : 'Polyline';  ItemCode : 42; ItemActive : True),
  (Item : 'Spray';     ItemCode : 43; ItemActive : True),
  (Item : 'Dragstamp'; ItemCode : 44; ItemActive : True),
  (Item : ''; ItemCode : 0; ItemActive : False),
  (Item : ''; ItemCode : 0; ItemActive : False),
  (Item : ''; ItemCode : 0; ItemActive : False),
  (Item : ''; ItemCode : 0; ItemActive : False),
  (Item : ''; ItemCode : 0; ItemActive : False),
  (Item : ''; ItemCode : 0; ItemActive : False),
  (Item : ''; ItemCode : 0; ItemActive : False),
  (Item : ''; ItemCode : 0; ItemActive : False),
  (Item : ''; ItemCode : 0; ItemActive : False),
  (Item : ''; ItemCode : 0; ItemActive : False),
  (Item : ''; ItemCode : 0; ItemActive : False),
  (Item : ''; ItemCode : 0; ItemActive : False),
  (Item : ''; ItemCode : 0; ItemActive : False),
  (Item : ''; ItemCode : 0; ItemActive : False),
  (Item : ''; ItemCode : 0; ItemActive : False))),
  (XStart    : 179; XEnd      : 215;
 Title     : 'Text';
 MenuSize  : 11;
 Imageptr  : NIL;
 Active    : True;
 Choices   : 5; ItemList  :
 ((Item : 'Load Font';     ItemCode : 51; ItemActive : True),
  (Item : 'Place Text';    ItemCode : 52; ItemActive : True),
                       (Item : 'Set Direction'; ItemCode : 53; ItemActive : True),
                       (Item : 'Dropshadow';    ItemCode : 54; ItemActive : True),
                       (Item : 'Point Size';    ItemCode : 55; ItemActive : True),
                       (Item : ''; ItemCode : 0; ItemActive : False),
                       (Item : ''; ItemCode : 0; ItemActive : False),
                       (Item : ''; ItemCode : 0; ItemActive : False),
                       (Item : ''; ItemCode : 0; ItemActive : False),
                       (Item : ''; ItemCode : 0; ItemActive : False),
                       (Item : ''; ItemCode : 0; ItemActive : False),
                       (Item : ''; ItemCode : 0; ItemActive : False),
                       (Item : ''; ItemCode : 0; ItemActive : False),
                       (Item : ''; ItemCode : 0; ItemActive : False),
                       (Item : ''; ItemCode : 0; ItemActive : False),
                       (Item : ''; ItemCode : 0; ItemActive : False),
                       (Item : ''; ItemCode : 0; ItemActive : False),
                       (Item : ''; ItemCode : 0; ItemActive : False),
                       (Item : ''; ItemCode : 0; ItemActive : False))),
                       (XStart    : 15; XEnd      : 55;
                      Title     : '';
                      MenuSize  : 0;
                      Imageptr  : NIL;
                      Active    : False;
                      Choices   : 0; ItemList  :
                      ((Item : ''; ItemCode : 0; ItemActive : False),
                       (Item : ''; ItemCode : 0; ItemActive : False),
                       (Item : ''; ItemCode : 0; ItemActive : False),
                       (Item : ''; ItemCode : 0; ItemActive : False),
                       (Item : ''; ItemCode : 0; ItemActive : False),
                       (Item : ''; ItemCode : 0; ItemActive : False),
                       (Item : ''; ItemCode : 0; ItemActive : False),
                       (Item : ''; ItemCode : 0; ItemActive : False),
                       (Item : ''; ItemCode : 0; ItemActive : False),
                       (Item : ''; ItemCode : 0; ItemActive : False),
                       (Item : ''; ItemCode : 0; ItemActive : False),
                       (Item : ''; ItemCode : 0; ItemActive : False),
                       (Item : ''; ItemCode : 0; ItemActive : False),
                       (Item : ''; ItemCode : 0; ItemActive : False),
                       (Item : ''; ItemCode : 0; ItemActive : False),
                       (Item : ''; ItemCode : 0; ItemActive : False),
                       (Item : ''; ItemCode : 0; ItemActive : False),
                       (Item : ''; ItemCode : 0; ItemActive : False),
                       (Item : ''; ItemCode : 0; ItemActive : False))),
                       (XStart    : 15; XEnd      : 55;
                      Title     : '';
                      MenuSize  : 0;
                      Imageptr  : NIL;
                      Active    : False;
                      Choices   : 0; ItemList  :
                      ((Item : ''; ItemCode : 0; ItemActive : False),
                       (Item : ''; ItemCode : 0; ItemActive : False),
                       (Item : ''; ItemCode : 0; ItemActive : False),
                       (Item : ''; ItemCode : 0; ItemActive : False),
                       (Item : ''; ItemCode : 0; ItemActive : False),
                       (Item : ''; ItemCode : 0; ItemActive : False),
                       (Item : ''; ItemCode : 0; ItemActive : False),
                       (Item : ''; ItemCode : 0; ItemActive : False),
                       (Item : ''; ItemCode : 0; ItemActive : False),
                       (Item : ''; ItemCode : 0; ItemActive : False),
             (Item : ''; ItemCode : 0; ItemActive : False),
             (Item : ''; ItemCode : 0; ItemActive : False),
             (Item : ''; ItemCode : 0; ItemActive : False),
             (Item : ''; ItemCode : 0; ItemActive : False),
             (Item : ''; ItemCode : 0; ItemActive : False),
             (Item : ''; ItemCode : 0; ItemActive : False),
             (Item : ''; ItemCode : 0; ItemActive : False),
             (Item : ''; ItemCode : 0; ItemActive : False),
             (Item : ''; ItemCode : 0; ItemActive : False))),
             (XStart    : 15; XEnd      : 55;
            Title     : '';
            MenuSize  : 0;
            Imageptr  : NIL;
            Active    : False;
            Choices   : 0; ItemList  :
            ((Item : ''; ItemCode : 0; ItemActive : False),
             (Item : ''; ItemCode : 0; ItemActive : False),
             (Item : ''; ItemCode : 0; ItemActive : False),
             (Item : ''; ItemCode : 0; ItemActive : False),
             (Item : ''; ItemCode : 0; ItemActive : False),
             (Item : ''; ItemCode : 0; ItemActive : False),
             (Item : ''; ItemCode : 0; ItemActive : False),
             (Item : ''; ItemCode : 0; ItemActive : False),
             (Item : ''; ItemCode : 0; ItemActive : False),
             (Item : ''; ItemCode : 0; ItemActive : False),
             (Item : ''; ItemCode : 0; ItemActive : False),
             (Item : ''; ItemCode : 0; ItemActive : False),
             (Item : ''; ItemCode : 0; ItemActive : False),
             (Item : ''; ItemCode : 0; ItemActive : False),
             (Item : ''; ItemCode : 0; ItemActive : False),
             (Item : ''; ItemCode : 0; ItemActive : False),
             (Item : ''; ItemCode : 0; ItemActive : False),
             (Item : ''; ItemCode : 0; ItemActive : False),
             (Item : ''; ItemCode : 0; ItemActive : False))),
            (XStart    : 15; XEnd      : 55;
            Title     : '';
            MenuSize  : 0;
            Imageptr  : NIL;
            Active    : False;
            Choices   : 0; ItemList  :
            ((Item : ''; ItemCode : 0; ItemActive : False),
             (Item : ''; ItemCode : 0; ItemActive : False),
             (Item : ''; ItemCode : 0; ItemActive : False),
             (Item : ''; ItemCode : 0; ItemActive : False),
             (Item : ''; ItemCode : 0; ItemActive : False),
             (Item : ''; ItemCode : 0; ItemActive : False),
             (Item : ''; ItemCode : 0; ItemActive : False),
             (Item : ''; ItemCode : 0; ItemActive : False),
             (Item : ''; ItemCode : 0; ItemActive : False),
             (Item : ''; ItemCode : 0; ItemActive : False),
             (Item : ''; ItemCode : 0; ItemActive : False),
             (Item : ''; ItemCode : 0; ItemActive : False),
             (Item : ''; ItemCode : 0; ItemActive : False),
             (Item : ''; ItemCode : 0; ItemActive : False),
             (Item : ''; ItemCode : 0; ItemActive : False),
             (Item : ''; ItemCode : 0; ItemActive : False),
             (Item : ''; ItemCode : 0; ItemActive : False),
             (Item : ''; ItemCode : 0; ItemActive : False),
 (Item : ''; ItemCode : 0; ItemActive : False))),
(XStart    : 15; XEnd      : 55;
Title     : '';
MenuSize  : 0;
Imageptr  : NIL;
Active    : False;
Choices   : 0; ItemList  :
((Item : ''; ItemCode : 0; ItemActive : False),
 (Item : ''; ItemCode : 0; ItemActive : False),
 (Item : ''; ItemCode : 0; ItemActive : False),
 (Item : ''; ItemCode : 0; ItemActive : False),
 (Item : ''; ItemCode : 0; ItemActive : False),
 (Item : ''; ItemCode : 0; ItemActive : False),
 (Item : ''; ItemCode : 0; ItemActive : False),
 (Item : ''; ItemCode : 0; ItemActive : False),
 (Item : ''; ItemCode : 0; ItemActive : False),
 (Item : ''; ItemCode : 0; ItemActive : False),
 (Item : ''; ItemCode : 0; ItemActive : False),
 (Item : ''; ItemCode : 0; ItemActive : False),
 (Item : ''; ItemCode : 0; ItemActive : False),
 (Item : ''; ItemCode : 0; ItemActive : False),
 (Item : ''; ItemCode : 0; ItemActive : False),
 (Item : ''; ItemCode : 0; ItemActive : False),
 (Item : ''; ItemCode : 0; ItemActive : False),
 (Item : ''; ItemCode : 0; ItemActive : False),
 (Item : ''; ItemCode : 0; ItemActive : False))),
(XStart    : 15; XEnd      : 55;
Title     : '';
MenuSize  : 0;
Imageptr  : NIL;
Active    : False;
Choices   : 0; ItemList  :
((Item : ''; ItemCode : 0; ItemActive : False),
 (Item : ''; ItemCode : 0; ItemActive : False),
 (Item : ''; ItemCode : 0; ItemActive : False),
 (Item : ''; ItemCode : 0; ItemActive : False),
 (Item : ''; ItemCode : 0; ItemActive : False),
 (Item : ''; ItemCode : 0; ItemActive : False),
 (Item : ''; ItemCode : 0; ItemActive : False),
 (Item : ''; ItemCode : 0; ItemActive : False),
 (Item : ''; ItemCode : 0; ItemActive : False),
 (Item : ''; ItemCode : 0; ItemActive : False),
 (Item : ''; ItemCode : 0; ItemActive : False),
 (Item : ''; ItemCode : 0; ItemActive : False),
 (Item : ''; ItemCode : 0; ItemActive : False),
 (Item : ''; ItemCode : 0; ItemActive : False),
 (Item : ''; ItemCode : 0; ItemActive : False),
 (Item : ''; ItemCode : 0; ItemActive : False),
 (Item : ''; ItemCode : 0; ItemActive : False),
 (Item : ''; ItemCode : 0; ItemActive : False),
 (Item : ''; ItemCode : 0; ItemActive : False))),
(XStart    : 15; XEnd      : 55;
Title     : '';
MenuSize  : 0;
Imageptr  : NIL;
Active    : False;
Choices   : 0; ItemList  :
((Item : ''; ItemCode : 0; ItemActive : False),
         (Item : ''; ItemCode : 0; ItemActive : False),
         (Item : ''; ItemCode : 0; ItemActive : False),
         (Item : ''; ItemCode : 0; ItemActive : False),
         (Item : ''; ItemCode : 0; ItemActive : False),
         (Item : ''; ItemCode : 0; ItemActive : False),
         (Item : ''; ItemCode : 0; ItemActive : False),
         (Item : ''; ItemCode : 0; ItemActive : False),
         (Item : ''; ItemCode : 0; ItemActive : False),
         (Item : ''; ItemCode : 0; ItemActive : False),
         (Item : ''; ItemCode : 0; ItemActive : False),
         (Item : ''; ItemCode : 0; ItemActive : False),
         (Item : ''; ItemCode : 0; ItemActive : False),
         (Item : ''; ItemCode : 0; ItemActive : False),
         (Item : ''; ItemCode : 0; ItemActive : False),
         (Item : ''; ItemCode : 0; ItemActive : False),
         (Item : ''; ItemCode : 0; ItemActive : False),
         (Item : ''; ItemCode : 0; ItemActive : False),
         (Item : ''; ItemCode : 0; ItemActive : False))),
        (XStart    : 15; XEnd      : 55;
        Title     : '';
        MenuSize  : 0;
        Imageptr  : NIL;
        Active    : False;
        Choices   : 0; ItemList  :
        ((Item : ''; ItemCode : 0; ItemActive : False),
         (Item : ''; ItemCode : 0; ItemActive : False),
         (Item : ''; ItemCode : 0; ItemActive : False),
         (Item : ''; ItemCode : 0; ItemActive : False),
         (Item : ''; ItemCode : 0; ItemActive : False),
         (Item : ''; ItemCode : 0; ItemActive : False),
         (Item : ''; ItemCode : 0; ItemActive : False),
         (Item : ''; ItemCode : 0; ItemActive : False),
         (Item : ''; ItemCode : 0; ItemActive : False),
         (Item : ''; ItemCode : 0; ItemActive : False),
         (Item : ''; ItemCode : 0; ItemActive : False),
         (Item : ''; ItemCode : 0; ItemActive : False),
         (Item : ''; ItemCode : 0; ItemActive : False),
         (Item : ''; ItemCode : 0; ItemActive : False),
         (Item : ''; ItemCode : 0; ItemActive : False),
         (Item : ''; ItemCode : 0; ItemActive : False),
         (Item : ''; ItemCode : 0; ItemActive : False),
         (Item : ''; ItemCode : 0; ItemActive : False),
         (Item : ''; ItemCode : 0; ItemActive : False))),
        (XStart    : 15; XEnd      : 55;
        Title     : '';
        MenuSize  : 0;
        Imageptr  : NIL;
        Active    : False;
        Choices   : 0; ItemList  :
        ((Item : ''; ItemCode : 0; ItemActive : False),
         (Item : ''; ItemCode : 0; ItemActive : False),
         (Item : ''; ItemCode : 0; ItemActive : False),
         (Item : ''; ItemCode : 0; ItemActive : False),
         (Item : ''; ItemCode : 0; ItemActive : False),
         (Item : ''; ItemCode : 0; ItemActive : False),
         (Item : ''; ItemCode : 0; ItemActive : False),
         (Item : ''; ItemCode : 0; ItemActive : False),
         (Item : ''; ItemCode : 0; ItemActive : False),
                        (Item : ''; ItemCode : 0; ItemActive : False),
                        (Item : ''; ItemCode : 0; ItemActive : False),
                        (Item : ''; ItemCode : 0; ItemActive : False),
                        (Item : ''; ItemCode : 0; ItemActive : False),
                        (Item : ''; ItemCode : 0; ItemActive : False),
                        (Item : ''; ItemCode : 0; ItemActive : False),
                        (Item : ''; ItemCode : 0; ItemActive : False),
                        (Item : ''; ItemCode : 0; ItemActive : False),
                        (Item : ''; ItemCode : 0; ItemActive : False),
                        (Item : ''; ItemCode : 0; ItemActive : False))));
```

Making Menus

All the data structures and subprograms it takes to use the menuing system are gathered together into a unit, **PullDown**. The heart of **PullDown** is the menuing routine itself, **Menu**. All the other functions and procedures in the unit support **Menu** somehow.

SetupMenu creates the menu bar across the top of the screen. This consists of drawing the white bar using the **Bar** procedure, then displaying the menu titles within their **XStart**, **XEnd** ranges across the bar. It also draws the small figure called the *amulet* at the far left corner of the menu bar. In the Macintosh, the amulet is the familiar Apple logo with the bite taken out of it; in other systems like GEM and Microsoft *Windows* the amulet is usually a small box with lines in it, as I have done here. The purpose of the amulet is to give a point on the menu that selects something that is not a menu. What this might be is up to you. When the user clicks on the amulet, a Boolean flag named **Amulet** is returned to the calling logic, which can then do what it must. A click on the amulet might trigger termination of the application (as it does in Microsoft *Windows*) or the forcing of the application to a background task from the foreground in a multitasking environment.

The best way to understand how menus are created and displayed is to follow the Pascal code of **Menu** through an invocation. The menu bar across the top of the screen is 12 pixels high. Any click on the mouse buttons when the mouse pointer Y value is less than 13 pixels from the top of the screen means that the user has clicked on the menu bar. An application should give control to **Menu** whenever it detects such a click.

From the top, then: **Menu**'s first task is to save the current draw color in a temporary variable called **SaveColor**. **Menu** uses only white, for simplicity's sake, but the application may be drawing in some other color when **Menu** is called.

That done, **Menu** polls the mouse to find out where the pointer is within the menu bar. The first check is for a click on the amulet. If the click was on the amulet, the **Amulet** parameter is set to **True** and **Menu** terminates. If the amulet was not the target of the user's click, **Menu** begins scanning the 13 menus in the **CurrentMenu** parameter to find out if the pointer lies within the title range for any active menu. This is done by testing the pointer X position against the **XStart** and **XEnd** fields in each menu in turn, starting with the leftmost menu. The scan continues until it finds an active menu, or until it finds a *null* menu; that is, a menu whose title is an empty string. By convention, all defined menus are grouped toward the left. Totally unused and empty menus should not be left within a group of defined menus; keep them at the end of the **MenuDesc** array. A defined menu coming *after* a null menu will not be detected in this initial scan.

If the pointer is not within the title range of an active menu, **Menu** terminates. Otherwise, the menu box is pulled down so that the user can select one of the menu's items.

The dimensions of the menu box enclosing a menu's list of items is not kept anywhere within the **MenuDesc** data structure. The width and height of the box is calculated each time the menu is pulled down. This is not difficult and it can be done quickly. The list of items is scanned to find the longest item title string. The width of the menu box is then calculated based on the length of this longest string, with a little margin space on either side of the string for clarity. The height of the menu box is easily calculated, since the entire system assumes the BGI 8 × 8 font, making each item's height exactly 12 pixels (8 for the font and four for breathing room above and below the characters themselves). The upper left corner coordinates are placed in variables **M1X** and **M1Y** and the left right corner coordinates are placed in variables **M2X** and **M2Y**. That done, we now know exactly what portion of the screen the menu box will cover when pulled down. The size in bytes that this region occupies is calculated using the **ImageSize** BGI function and saved in a **MenuRec** field named **MenuSize**.

We have to save whatever is underneath that part of the screen. This is done by allocating space on the heap for a bitmap (using the **MenuSize** figure). We then save the screen area under the menu box to the heap as a bitmap by using the BGI procedure **GetImage**.

Once the area under the menu box is safely stored on the heap, we can start building the menu box itself. The first thing to go up is the bounce bar. It is drawn using the BGI **Bar** procedure, and immediately stored on the heap using **GetImage**. We will frequently need it again.

The fastest way to show a menu on the screen is to move a bitmap containing the menu box pattern from the heap directly to screen memory using **PutImage**. The menu box pattern has to first get onto the heap somehow before we can do that. The way **Menu** was designed, the *first* time a menu box is shown to the user, it is drawn right on the screen using the various BGI procedures. Once drawn, it is saved out to the heap, and every time after that it is "flashed" to the screen using **PutImage**, which is several times faster than drawing the menu box piecemeal.

How does **Menu** tell whether a given menu box has been drawn once before? In the **MenuRec** record is a pointer field called **ImagePtr**. A typed constant can only initialize a pointer field to **Nil**, and **Nil** is the value **ImagePtr** has until its menu box is drawn onto the screen. When the menu box pattern is moved to the heap, **ImagePtr** is made to point to the pattern on the heap using **GetMem**. After that, **ImagePtr** will no longer be equal to **Nil**. So when **Menu** sees a menu's **ImagePtr** pointer equal to **Nil**, it knows that the menu has not been shown before and must be drawn from scratch on the screen.

In connection with this, note that changing the active/inactive status of an item or menu alters the pattern of the menu box that contains that item. The procedures that change the active/inactive status of an item or menu (**ActivateItem**, **DeactivateItem**,

ActivateMenu, and **DeactivateMenu**) force **ImagePtr** to **Nil**, which in turn forces **Menu** to draw that menu from scratch the next time the user pulls it down.

The menu box border is drawn with the BGI **Rectangle** procedure, and the item titles are drawn using **OutTextXY**. Note that the top item is drawn in black rather than white. Remember also that we have already drawn the bounce bar on the screen, and the top item in a menu is drawn on top of the bounce bar. After the first item is drawn, the draw color is returned to white using the BGI **SetColor** procedure. When the entire menu box pattern has been drawn, it is saved to the heap using **GetImage**.

At this point, the menu box pattern exists on the heap, whether it was just drawn and stored there or stored there the last time the menu was pulled down. **PutImage** is thus used to bring the menu box pattern from the heap to the display.

With the menu down and on display, the next step is to *bounce* the bounce bar up and down the list of items following the motion of the mouse pointer. This is done with a **REPEAT..UNTIL** loop that repeatedly samples the mouse pointer using **PollMouse**. The loop keeps track of where the mouse pointer is in relation to the bounce bar, and when the mouse pointer moves up or down beyond the limits of the bounce bar, the bounce bar is bounced up or down to follow the mouse pointer. Of course, checks are made to ensure that the bounce bar is not bounced off the top or the bottom of the menu box.

Bouncing the bar is done using the **XOROpt** parameter to **PutImage**. The bounce bar pattern is a plain white bar stored on the heap. The top item in the menu, if you recall, was written with black letters on the original white bounce bar. If the plain bounce bar image on the heap is XORed over the bounce bar and its black letters, the black letters will become white, and the white bounce bar will disappear.

On the other hand, if the plain white bounce bar is XORed over white letters against black screen, the result will be a white bar with black letters. Using **PutImage** and **XOROpt**, the bounce bar is erased from its original position and written again either one position up or one position down. The upper and lower bounds of the bar are changed accordingly, as is a "pick" value that started out as 0 and indicates which of the menu items the bounce bar is currently highlighting.

As the user moves the mouse pointer up and down while holding the left mouse button depressed, the bounce bar follows, highlighting one menu item or another. All this time the loop code waits for one of the following things to happen:

1. The user presses the right mouse button. This indicates that the user has had a change of heart and no longer wishes to select a menu item. It's like pressing the Esc key; the idea is to put the menu away without selecting anything. The menu box is erased (we say, "put away") by writing the previously saved portion of the screen back over it, (this happens within the **RestoreUnderMenuBox** local procedure) and **Menu** returns a 0 code to the calling logic in the **ReturnCode** parameter.
2. The user moves the mouse pointer outside the bounds of the menu box. This is equivalent to pressing the right mouse button. The menu box is put away and a 0 code is returned by Menu to the calling logic.
3. The user *releases* the left mouse button while the bounce bar is highlighting an active

item. This indicates selection of that item. The item's code is passed back to the calling logic in **ReturnCode**, and the menu is put away.

If the user releases the left button while the pointer is pointing to an inactive menu item, a 0 code will be returned. Note that the bounce bar does not highlight inactive menu items; they remain black and without a visible item title.

```
{--------------------------------------------------------------}
{                          PULLDOWN                            }
{                                                              }
{              Graphics pull-down menuing system               }
{                                                              }
{                              by Jeff Duntemann               }
{                              Turbo Pascal V5.0               }
{                              Last update 7/24/88             }
{                                                              }
{                                                              }
{                                                              }
{                                                              }
{--------------------------------------------------------------}

UNIT PullDown;

INTERFACE

USES DOS,Graph,Crt,Mouse;  { Mouse is described in Section 17 }

TYPE
  String15   = String[15];

  ItemRec    = RECORD
                 Item       : String15;  { Title of item }
                 ItemCode   : Byte;      { Code number of item }
                 ItemActive : Boolean    { True if item is active }
               END;

  MenuRec    = RECORD
                 XStart,XEnd : Word;      { Pixel offset along menu bar }
                 Title       : String15;  { Menu title }
                 MenuSize    : Word;      { Size of menu image on heap }
                 Imageptr    : Pointer;   { Points to menu image on heap }
                 Active      : Boolean;   { True if menu is active }
                 Choices     : Byte;      { Number of items in menu }
                 ItemList    : ARRAY[0..18] OF ItemRec  { The items }
               END;

  MenuDesc   = ARRAY[0..12] OF MenuRec;  { Up to 13 items along menu bar }


{->>>>ActivateMenu<<<<------------------------------------------}
{                                                               }
{ Filename : PULLDOWN.PAS -- Last Modified 12/25/87             }
{                                                               }
{ This routine makes the menu specified by MenuNumber active,   }
{ regardless of whether it was active or inactive at            }
{ invocation.  ImagePtr is set to NIL so that the menu will be  }
{ redrawn the next time it is pulled down.                 }
{                                                          }
{ Types MenuRec, ChoiceRec, MenuDesc and String15 must be  }
{   predefined.                                            }
{----------------------------------------------------------}

PROCEDURE ActivateMenu(VAR CurrentMenu : MenuDesc;
                           MenuNumber  : Byte);


{->>>>DeactivateMenu<<<<-----------------------------------}
{                                                          }
{ Filename : PULLDOWN.PAS -- Last Modified 12/25/87        }
{                                                          }
{ This routine makes the menu specified by MenuNumber      }
{ inactive, regardless of whether it was active or inactive at }
{ invocation.  ImagePtr is set to NIL so that the menu will be }
{ redrawn the next time it is pulled down.                 }
{                                                          }
{ Types MenuRec, ChoiceRec, MenuDesc and String15 must be  }
{   predefined.                                            }
{----------------------------------------------------------}

PROCEDURE DeactivateMenu(VAR CurrentMenu : MenuDesc;
                             MenuNumber  : Byte);


{->>>>ActivateItem<<<<-------------------------------------}
{                                                          }
{ Filename : PULLDOWN.PAS -- Last Modified 12/25/87        }
{                                                          }
{ This routine sets the item whose code is given in Code to }
{ active, regardless of the state of the item at invocation. }
{ ImagePtr is set to NIL so that the menu will be redrawn  }
{ the next time it is pulled down.                         }
{                                                          }
{ Types MenuRec, ChoiceRec, MenuDesc and String15 must be  }
{   predefined.                                            }
{----------------------------------------------------------}

PROCEDURE ActivateItem(VAR CurrentMenu : MenuDesc;
                           Code        : Byte);


{->>>>DeactivateItem<<<<-----------------------------------}
{                                                          }
{ Filename : PULLDOWN.PAS -- Last Modified 12/25/87        }
{                                                          }
{ This routine sets the item whose code is given in Code to }
{ inactive, regardless of the state of the item at invocation. }
{ ImagePtr is set to NIL so that the menu will be redrawn  }
{ the next time it is pulled down.                         }
{                                                          }
{ Types MenuRec, ChoiceRec, MenuDesc and String15 must be  }
{   predefined.                                            }
{----------------------------------------------------------}

PROCEDURE DeactivateItem(VAR CurrentMenu : MenuDesc;
                              Code            : Byte);


{->>>>InvalidMenu<<<<--------------------------------------------}
{                                                                }
{ Filename : PULLDOWN.PAS -- Last Modified 12/25/87              }
{                                                                }
{ This function checks for duplicate item codes within the       }
{ menu array passed in CurrentMenu.  The menuing system always }
{ assumes that every menu item has a unique code.  Run this     }
{ function on any menu array you intend to use and abort if a   }
{ duplicate code is detected.                                    }
{                                                                }
{ Types MenuRec, ChoiceRec, MenuDesc and String15 must be       }
{   predefined.                                                  }
{----------------------------------------------------------------}

FUNCTION InvalidMenu(CurrentMenu : MenuDesc;
                     VAR BadCode : Byte) : Boolean;



{->>>>SetupMenu<<<<----------------------------------------------}
{                                                                }
{ Filename : PULLDOWN.PAS -- Last Modified 12/25/87              }
{                                                                }
{ This routine does the initial display of the menu bar, menu   }
{ titles, and the menu bar amulet.                               }
{                                                                }
{ Types MenuRec, ChoiceRec, MenuDesc and String15 must be       }
{   predefined.                                                  }
{----------------------------------------------------------------}

PROCEDURE SetupMenu(CurrentMenu : MenuDesc);



{->>>>Menu<<<<---------------------------------------------------}
{                                                                }
{ Filename : PULLDOWN.PAS -- Last Modified 12/25/87              }
{                                                                }
{ This is the main menuing routine.  It requires that both      }
{ InvalidMenu and SetupMenu be run before it.  It directly      }
{ samples the mouse pointer position and decides which menu     }
{ within the menu bar has been selected.  It then allows the    }
{ user to bounce the menu bar up and down within the menu       }
{ until an item is chosen or the right button is pressed or     }
{ the pointer is moved out of the pulled-down menu.  The code   }
{ of the chosen item is returned in ReturnCode.  If no item is }
{ chosen, ReturnCode comes returns a 0.  The returned code is   }
{ within the range 0-255.                                        }
{                                                                }
{ Menu is responsible for drawing pull-down menus and storing   }
{ them on the heap so that once drawn a menu does not need to   }
{ be drawn again until it is changed somehow, typically by      }
{ deactivating or reactivating an item.                          }
{                                                                }
{ Types MenuRec, ChoiceRec, MenuDesc and String15 must be       }
{   predefined.                                              }
{-----------------------------------------------------------}

PROCEDURE Menu(CurrentMenu    : MenuDesc;
               VAR ReturnCode : Word;
               VAR Amulet     : Boolean);



IMPLEMENTATION


PROCEDURE ChangeItemStatus(VAR CurrentMenu : MenuDesc;
                               Code        : Byte;
                               ToActive    : Boolean);

VAR
  I          : Byte;
  MenuNumber : Byte;
  ItemFound  : Boolean;

BEGIN
  MenuNumber := 0; ItemFound := False;
  REPEAT
    WITH CurrentMenu[MenuNumber] DO
      BEGIN
        I := 0;
        REPEAT     { Here we scan menu items to find the right one }
          IF ItemList[I].ItemCode = Code THEN  { We found it ! }
            BEGIN
              ItemList[I].ItemActive := ToActive;  { Mark item }
              ItemFound := True;
              { Since we've changed the information in a menu, we must      }
              {  remove any menu image from storage on the heap, and force  }
              {  the code to redraw the menu the next time it's pulled down: }
              IF ImagePtr <> NIL THEN    { If there's an image on the heap }
                BEGIN
                  FreeMem(ImagePtr,MenuSize);  { Deallocate the heap image }
                  ImagePtr := NIL              { Make pointer NIL again }
                END;
            END
          ELSE
            Inc(I)
        UNTIL ItemFound OR (I > Choices)
      END;
    Inc(MenuNumber)
  UNTIL ItemFound OR (MenuNumber > 12);
END;


{-------------------------------------------------------------------}
{  IMPLEMENTATION Definitions above this bar are PRIVATE to the unit. }
{-------------------------------------------------------------------}


PROCEDURE ActivateMenu(VAR CurrentMenu : MenuDesc;
                           MenuNumber      : Byte);

```

```
BEGIN
  WITH CurrentMenu[MenuNumber] DO
    BEGIN
      ImagePtr := NIL;
      Active   := True
    END
END;


PROCEDURE DeactivateMenu(VAR CurrentMenu : MenuDesc;
                             MenuNumber      : Byte);

BEGIN
  WITH CurrentMenu[MenuNumber] DO
    BEGIN
      ImagePtr := NIL;
      Active   := False
    END
END;




PROCEDURE ActivateItem(VAR CurrentMenu : MenuDesc;
                           Code            : Byte);

BEGIN
  ChangeItemStatus(CurrentMenu,Code,True)
END;


PROCEDURE DeactivateItem(VAR CurrentMenu : MenuDesc;
                             Code            : Byte);

BEGIN
  ChangeItemStatus(CurrentMenu,Code,False)
END;




FUNCTION InvalidMenu(CurrentMenu : MenuDesc;
                     VAR BadCode : Byte) : Boolean;

VAR
  I,J            : Word;
  CmdSet         : SET OF Byte;
  DuplicateFound : Boolean;

BEGIN
  DuplicateFound := False;
  CmdSet := [];  { Start out with the empty set }
  FOR I := 0 TO 12 DO      { Check each menu }
    WITH CurrentMenu[I] DO
      BEGIN
        J := 0;  { Reset item counter to 0 for each new menu }
        REPEAT   { Here we scan menu items to check each one }
          IF ItemList[J].ItemCode > 0 THEN
```

```
            IF ItemList[J].ItemCode IN CmdSet THEN
              BEGIN
                DuplicateFound := True;         { Flag duplicate }
                BadCode := ItemList[J].ItemCode { Return dupe in BADCODE }
              END
            ELSE
              BEGIN
                { Add item's command code to the set: }
                CmdSet := CmdSet + [ItemList[J].ItemCode];
                Inc(J)
              END
          ELSE Inc(J)
        UNTIL (J > Choices) OR DuplicateFound
      END;
  InvalidMenu := DuplicateFound
END;



PROCEDURE SetupMenu(CurrentMenu : MenuDesc);

VAR
  I,DrawX,DrawY : Word;

BEGIN
  { Show bar and amulet: }
  SetFillStyle(SolidFill,White); Bar(0,0,GetMaxX,11);
  SetColor(0); Rectangle(2,1,12,9);
  FOR I := 3 TO 8 DO IF Odd(I) THEN Line(4,I,10,I);

  { Display menu titles in bar: }
  DrawX := CurrentMenu[0].XStart; DrawY := 2; I := 0;
  REPEAT
    OutTextXY(DrawX,DrawY,CurrentMenu[I].Title);
    Inc(I);
    DrawX := CurrentMenu[I].XStart;
  UNTIL (Length(CurrentMenu[I].Title) = 0) OR (I > 13);
END;


PROCEDURE Menu(CurrentMenu     : MenuDesc;
               VAR ReturnCode : Word;
               VAR Amulet     : Boolean);

VAR
  PointerX,PointerY : Word;      { Current position of mouse pointer }
  Left,Center,Right : Boolean;   { Current state of mouse buttons }
  I,J               : Integer;
  MenuWidth         : Integer;   { Width in pixels of target menu }
  M1X,M1Y,M2X,M2Y   : Integer;   { Coordinates of menu box }
  FoundMenu         : Boolean;
  SaveColor         : Integer;   { Holds caller's draw color }
  UnderMenu         : Pointer;   { Points to saved screen area }
  BounceBar         : Pointer;   { Points to bounce bar pattern }
  Pick              : Word;      { Number of item under bounce bar }
  UpperBound,
    LowerBound      : Integer;   { Current Y-limits of bounce bar }

```

```

PROCEDURE RestoreUnderMenuBox;

BEGIN
  PointerOff;
  PutImage(M1X,M1Y,UnderMenu^,NormalPut);
  PointerOn
END;


BEGIN
  Amulet := False;
  SaveColor := GetColor; SetColor(White);
  PollMouse(PointerX,PointerY,Left,Right,Center);
  { Check to see if the amulet is under mouse pointer: }
  IF (PointerX > 1) AND (PointerX < 13) AND
     (PointerY > 0) AND (PointerY < 10)
  THEN
    BEGIN
      Amulet := True;   { We've clicked on the amulet }
      SetColor(SaveColor);
      Exit              { THIS IS AN EXIT TO MENU! }
    END;
  { Now we find out which menu to pull down: }
  I := -1;
  REPEAT
    I := I + 1;
    IF (PointerX >= CurrentMenu[I].XStart) AND  { If pointer is in }
       (PointerX <= CurrentMenu[I].XEnd)   AND  { menu's range }
       CurrentMenu[I].Active                    { and menu is active }
    THEN FoundMenu := True ELSE FoundMenu := False;
  UNTIL FoundMenu OR                            { We hit an active menu }
        (Length(CurrentMenu[I].Title) = 0) OR   { We hit a null menu }
        (I > 13);                               { Only 13 menus max! }
  IF FoundMenu THEN  { Pull it down and pick! }
    BEGIN
      PointerOff;
      WITH CurrentMenu[I] DO   { We're only working with current menu now }
        BEGIN
          { Calc coordinates of the found menu box: }
          MenuWidth := 0;        { First we have to calc menu width : }
          FOR J := 0 TO Choices-1 DO  { Find longest item string }
            IF Length(ItemList[J].Item) > MenuWidth
              THEN MenuWidth := Length(ItemList[J].Item);
          MenuWidth := MenuWidth * 8; { We're using the 8 X 8 font }
          M1X := XStart; M1Y := 11;
          M2X := XStart+MenuWidth+6;
          M2Y := (Choices+1) * 12;
          MenuSize := ImageSize(M1X,M1Y,M2X,M2Y);

          { We must save the screen area beneath the menu box: }
          GetMem(UnderMenu,MenuSize);                { Allocate space on heap }
          GetImage(M1X,M1Y,M2X,M2Y,UnderMenu^);  { Save area out to heap  }

          { First we clear the menu box: }
          SetFillStyle(SolidFill,Black);
          Bar(M1X,M1Y,M2X,M2Y);

```

```
{ Here we create the bounce bar pattern on the heap: }
SetFillStyle(SolidFill,White);
GetMem(BounceBar,ImageSize(M1X+1,M1Y+1,M2X-1,M1Y+12));
Bar(M1X+1,M1Y+1,M2X-1,M1Y+12);
GetImage(M1X+1,M1Y+1,M2X-1,M1Y+12,BounceBar^);

{ If the menu has not yet been shown for the first time, or if    }
{   the active/inactive status of any menu item has changed since }
{   we last pulled it down, the image pointer is NIL and we must  }
{   draw it and then store it on the heap.  Any time AFTER the    }
{   first time it comes in from the heap with lightning speed...  }
IF ImagePtr = NIL THEN    { We must draw the menu }
  BEGIN
    Rectangle(M1X,M1Y,M2X,M2Y);  { Draw the menu box }
    { The first item must be drawn in black on the white bar: }
    SetColor(Black);
    IF ItemList[0].ItemActive THEN
      OutTextXY(XStart+3,14,ItemList[0].Item);
    SetColor(White);
    { Items after the first are drawn in white on black: }
    FOR J := 1 TO Choices-1 DO IF ItemList[J].ItemActive THEN
      OutTextXY(XStart+3,14+(J*12),ItemList[J].Item);
    { Now we allocate heap space and move image to heap }
    GetMem(ImagePtr,MenuSize);
    GetImage(M1X,M1Y,M2X,M2Y,ImagePtr^);
  END;

{ Bring the menu box image in from the heap: }
PutImage(M1X,M1Y,ImagePtr^,NormalPut);
PointerOn;  { We need the pointer on to bounce the bar }

{ Now we enter the "bounce loop" that moves the bounce bar  }
{  up and down the menu box, attached to the mouse pointer: }
UpperBound := 12; LowerBound := 24; Pick := 0;
REPEAT
  PollMouse(PointerX,PointerY,Left,Center,Right);
  { If the pointer leaves the menu box, it's an "escape" }
  {   identical in effect to pressing the right button:  }
  IF (PointerX < M1X) OR (PointerX > M2X) OR
     (PointerY > M2Y) THEN Right := True
  ELSE
    BEGIN
    IF PointerY < UpperBound THEN   { We bounce the bar UPWARD: }
      IF PointerY > 12 THEN   { If we're not above the top line }
        BEGIN
          PointerOff;
          { Erase bar at current position if item is active: }
          IF ItemList[Pick].ItemActive THEN
            PutImage(M1X+1,UpperBound,BounceBar^,XORPut);
          { Decrement bounds and pick number: }
          UpperBound := UpperBound - 12;
          LowerBound := LowerBound - 12;
          Pick := Pick - 1;
          { Show bar at new position if item is active: }
          IF ItemList[Pick].ItemActive THEN
            PutImage(M1X+1,UpperBound,BounceBar^,XORPut);
          PointerOn;
        END;
```

```
       IF PointerY > LowerBound THEN
        BEGIN
         PointerOff;
         { Erase bar at current position if item is active: }
         IF ItemList[Pick].ItemActive THEN
          PutImage(M1X+1,UpperBound,BounceBar^,XORPut);
         { Increment bounds and pick number: }
         UpperBound := UpperBound + 12;
         LowerBound := LowerBound + 12;
         Pick := Pick + 1;
         { Show bar at new position if item is active: }
         IF ItemList[Pick].ItemActive THEN
          PutImage(M1X+1,UpperBound,BounceBar^,XORPut);
         PointerOn;
        END;
       END;
     UNTIL (NOT Left) OR Right;
     RestoreUnderMenuBox;
     { Now we set up the function return code.  The right button  }
     { always indicates "escape;" i.e., 0; Take No Action.         }
     { Picking an inactive menu item also returns a 0.  An active }
     { item returns its item code as the function result. }
     IF Right THEN ReturnCode := 0
      ELSE IF ItemList[Pick].ItemActive THEN
          ReturnCode := ItemList[Pick].ItemCode
         ELSE ReturnCode := 0
    END;  { WITH statement }
   PointerOn;
  END;
 SetColor(SaveColor);   { Restore caller's drawing color }
END;

{ No initialization section...}

END.
```

A Graphics Scratchpad Application

To illustrate the **PullDown** unit, I have written a very simple graphics scratchpad application called **Scribble**. Between **PullDown** and **Scribble**, you have examples of the majority of the routines built into Turbo Pascal's **Graph** unit.

Scribble doesn't actually do very much aside from allow you to create freehand sketches on the screen. The majority of the menu items in the **DemoMenu** structure are dummies. All of the command structure is in place for you to expand **Scribble**, however, just by adding new case handlers to the central **Case** statement. It does show you how to initialize the BGI and enter graphics mode, check for BGI errors, and exit graphics mode courteously without abandoning the user in some anomalous state with a scrambled screen and no cursor.

An important point of interest is that the "cleanup" work of turning off the mouse driver and re-entering text mode are done from the main program's exit procedure. This ensures that, in the event of a runtime error of some sort, the system will return to text mode no matter where in the program the runtime error occurred. This is tested by deliberately staging a divide-by-zero error when the RENAME option in the FILES menu is selected. If **CloseGraph** had not be present in the exit procedure when that error is triggered, **Scribble** would return to DOS while remaining in text mode, leaving the user up the proverbial creek without a cursor.

```
{--------------------------------------------------------------}
{                          SCRIBBLE                            }
{                                                              }
{              Freehand graphics sketchpad program             }
{                                                              }
{                                by Jeff Duntemann             }
{                                Turbo Pascal V5.0             }
{                                Last update 7/24/88           }
{                                                              }
{                                                              }
{                                                              }
{                                                              }
{--------------------------------------------------------------}

PROGRAM Scribble;

USES DOS,Crt,Graph,Mouse,PullDown;

{$I DEMOMENU.DEF }  { This is a LARGE include file containing a }
                    { sample menu array of type MenuDesc }


VAR
  GraphDriver : Integer;
  GraphMode   : Integer;
  ErrorCode   : Integer;
  I           : Integer;
  R           : Real;
  M1,M2,M3,M4 : Word;
  ReturnCode  : Word;
  XText,YText : String;
  Mule        : String;
  PointerX,
    PointerY  : Word;
  Left,Center,
    Right     : Boolean;
  Amulet      : Boolean;   { True if amulet was clicked on within Menu }
  Quit        : Boolean;
  DuplicateCode : Byte;
  ExitSave    : Pointer;

VAR
  SavedColor : Word;
  Palette : PaletteType;
  Color   : Word;


{$F+} PROCEDURE ReturnToTextMode; {$F-}

BEGIN
```

```
  PointerOff;           { Turn off the mouse pointer }
  CloseGraph;           { Go back to text mode }
  ExitProc := ExitSave
END;


BEGIN
  ExitSave := ExitProc;            { This ensures that we will ALWAYS     }
  ExitProc := @ReturnToTextMode;   { re-enter text mode on return to DOS! }

  ClrScr;
  { Check the menu structure to be sure all codes are unique: }
  IF InvalidMenu(DemoMenu,DuplicateCode) THEN
    BEGIN
      Writeln('>>Halted for invalid menu: Duplicate code ',
              DuplicateCode);
      Halt(1)
    END;

  GraphDriver := Detect;  { Let the BGI determine what board we're using }
  InitGraph(GraphDriver,GraphMode,'');
  IF ErrorCode <> 0 THEN
    BEGIN
      Writeln('>>Halted on graphics error: ',GraphErrorMsg(ErrorCode));
      Halt(2)
    END;

  SetupMenu(DemoMenu);
  PointerOn;

  SetColor(Yellow); Quit := False;
  REPEAT
    PollMouse(PointerX,PointerY,Left,Center,Right);
    IF Left THEN
      BEGIN
        IF PointerY < 12 THEN { We're in the menu bar; call Menu: }
          BEGIN
            Menu(DemoMenu,ReturnCode,Amulet);
            { Update graphics CP to reflect motion while in Menu: }
            PollMouse(PointerX,PointerY,Left,Center,Right);
            MoveTo(PointerX,PointerY);

            { This CASE..OF statement parses menu items and takes action }
            IF ReturnCode <> 0 THEN
              CASE ReturnCode OF
                25 : Quit := True;
            { Here is where you parse out other codes returned from }
            { procedure Menu.  Put a CASE selector for each valid   }
            { code, and then implement some action for each CASE    }
            { selector. }
              END;  { CASE }
          END
        ELSE
          IF (GetX <> PointerX) OR (GetY <> PointerY) THEN
            BEGIN   { If we not in the menu bar, then we sketch: }
              PointerOff;
              LineTo(PointerX,PointerY);
              PointerOn
              END
          END
        ELSE MoveTo(PointerX,PointerY)
      UNTIL KeyPressed OR Quit;

      { Note that CloseGraph is executed from the MAIN exit procedure! }
    END.
```

23 Low Level System Hooks

Portability is a myth.

Programs written in ISO Standard Pascal are reasonably portable from one machine to another, largely because ISO Standard Pascal ignores the details of each machine. Programs written under Standard Pascal are akin to batch programs running on mainframe machines: no interaction with the user, and no interaction with peripherals other than **Input** and **Output**. Nothing in the Pascal standard allows the connection of a logical file to any physical file; so technically, ISO Standard Pascal can't handle disk file I/O at all, which makes portability a myth in any event.

In the real world, you cannot ignore the machine. In these late 1980s, there are only a handful of different computers that matter in the slightest: The PC, the Macintosh, the Atari ST, and (barely) the Amiga. The differences between architectures are more striking than their similarities, especially if, like many, you limit the list of important machines to the PC and Macintosh only.

So think for a minute, and answer me this: Could you write a serious application that could be compiled from identical source code on both the PC and the Mac? I'm laughing. You should be too.

My personal philosophy points in quite the other direction: You should take as much advantage of machine-specific features as you possibly can. People who buy the machines pay for those features, and they expect them to be used. Using those features will make your programs faster, easier to use, and easier to write. Tossing aside advantages like those is too high a price to pay for the ability to move a program painlessly from one machine to another.

23.1: THE ANATOMY OF A TURBO PASCAL PROGRAM

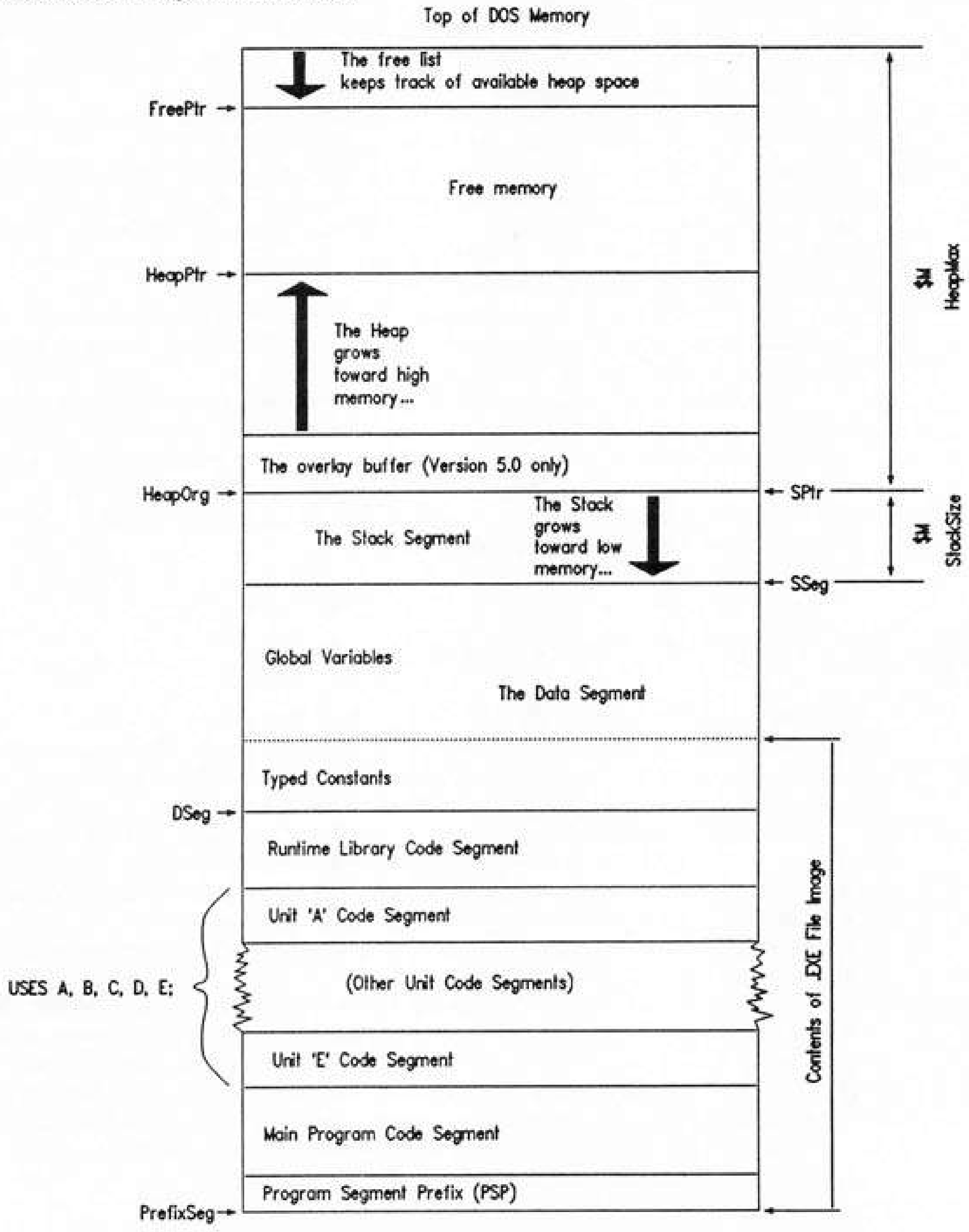
Hooking into the system first requires understanding the system. A good place to start is understanding the internal structure of a Turbo Pascal program. Things have changed considerably since Turbo Pascal 3.0, which was a .COM file having a single code segment. It's a more complicated world, but a far richer one.

First of all, starting with release 4.0, Turbo Pascal generates .EXE files instead of .COM files. An .EXE file can have many code segments rather than just one, and it also allows programs a lot more control over how much of system memory is allocated to the stack and the heap. A full explanation of what a .EXE file is (especially of the .EXE file's header) goes beyond the scope of this book, but I will try to get across what you'll need to know to make the best use of your PC.

Figure 23.1 is a memory diagram of a typical program generated by the Turbo Pascal compiler, after the program has been loaded into memory and begun running. The lowest 256 bytes of the running program are called the *program segment prefix* (PSP). It is initialized by DOS when DOS loads your program's .EXE file from disk and starts it running. The PSP contains a great many things, few of them useful or interesting. The second half of the PSP is used as the default Disk Transfer Area (DTA) which we discussed in Chapter 20. A predefined variable of type **Word**, **PrefixSeg**, contains the segment address of the PSP.

Figure 23.1

A Turbo Pascal Program in Memory


Top of DOS Memory
The free list keeps track of available heap space
FreePtr
Free memory
HeapPtr
The Heap grows toward high memory...
The overlay buffer (Version 5.0 only)
HeapOrg
SPtr
$M HeapMax
The Stack Segment
The Stack grows toward low memory...
SSeg
$M StackSize
Global Variables
The Data Segment
Typed Constants
DSeg
Runtime Library Code Segment
Unit 'A' Code Segment
USES A, B, C, D, E;
(Other Unit Code Segments)
Unit 'E' Code Segment
Main Program Code Segment
Program Segment Prefix (PSP)
PrefixSeg
Contents of .EXE File Image

Starting at offset $100 into the program is the code segment of the main program. It can be as large as 65,536 bytes. Following the main program's code segments are the code segments of any units used by the program, in *reverse* order of their declaration in the **USES** statement. In other words, given this **USES** statement:

```
USES DOS,Crt,Mouse,Pulldown;
```

the main program code segment would come first in memory, followed by code segments for units **Pulldown**, **Mouse**, **Crt**, and **DOS**, in that order. After units cited in the **USES** statement comes the code segment of the runtime library, also called the **System** unit. All of these units may be 64K in size, though few of them will ever be even close to that.

The code portion of the program ends after the **System** unit. The data segment begins at that point. The segment address of the start of the data segment can be returned with the **DSeg** function. The data segment begins with any typed constants that might exist in the main program or any of the units. Version 3.0 and earlier kept typed constants in the code segment, because only the code segment is saved to disk in a .COM file. An .EXE file can contain a "memory image" of any portion of the data segment, so typed constants are kept in the first part of the data segment, and that part of the data segment is saved to disk along with the several code segments as part of the .EXE file.

The data segment contains global variables for the main program and any units used by the program. Local variables are allocated on the stack when their procedures or functions are called, and therefore do not exist in the data segment.

Immediately above the data segment is the stack segment. The segment address of the stack segment may be returned as a **Word** value by using the built-in **SSeg** function. The stack segment address never changes, but the *stack pointer* provides an offset into the stack segment that indicates the last occupied location on the stack. This we often call "the top of the stack," even if the stack is often said to grow "downward." (Yes, it's a crazy business, this low-level stuff.)

The current value of the stack pointer may be returned as a **Word** value by using the built-in **SPtr** function. When the program first begins running, the stack pointer is set to 1 byte *past* the last byte in the stack segment. That way, when the first byte is pushed onto the stack, it will be pushed into the last byte in the stack segment. The stack pointer is then decremented so that it points to that byte.

As data is pushed onto the stack, the stack pointer moves downward, toward the data segment. How much room the stack segment has is governed by the **$M** compiler directive (see Section 20.10), but the default value is 16,384 bytes. Like all 8086 segments, the stack segment may be as large as 64K. To allow more than 16K for the stack segment, you must specify the stack size with **$M**.

Above the stack segment lies the heap. The heap is not a single segment, but is what we call a *long heap.* In other words, every pointer in a Turbo Pascal program is a full 32-bit address, allowing the heap to occupy all remaining system memory, right up to the 640K mark.

Under Turbo Pascal 5.0, the first chunk of heap may be allocated to the overlay buffer, if you specify that your program is to use overlays. The overlay buffer must be allocated before *anything* else is allocated on the heap. If the overlay buffer exists at all, it exists at the very start of the heap, immediately above the stack segment. For more on overlays, see Section 17.5.

How much space the heap occupies is govered by two values, not simply one:

- *Minimum heap size* (default: 0 KB) is the amount of memory that must be allocatable to the heap in order for the program to allow itself to run. In other words, once code and stack are allocated memory (and they get first grab) there must be enough memory remaining to satisfy the minimum heap size, or the program will abort to DOS without running. You may think of this as the amount of memory the heap *needs*.
- *Maximum heap size* (default: 655360 KB) is the amount of memory the program will allocate for itself *if the memory is available*. Think of this as the amount of memory the heap *wants*. The default request is all remaining memory, so in effect the heap will take anything not used by code and stack. This value may be reduced from 655360 KB to allow space in high memory for special purposes.

As with the stack, the size of the heap is dictated by the **$M** compiler directive. Both minimum and maximum heap size are specified by **$M**, as explained in Section 20.10.

The heap begins at a location contained in a predefined pointer variable **HeapOrg**. As new dynamic variables are allocated on the heap, they are allocated the first area of memory large enough to contain them. The end of the highest allocated memory block is indicated by another predefined pointer variable, **HeapPtr**. It is analogous to the stack pointer, in that it marks the boundary between allocated and totally free memory. However, unlike the stack, there may be *holes* of free memory beneath **HeapPtr**. These holes were left when the program disposed of a dynamic variable and marked the previous location of that variable as free memory.

Turbo Pascal keeps track of these holes with something called the *free list*, which is a stack-like list of descriptor records that begins at the very top of heap memory and grows downward toward allocated memory. Each descriptor record contains a pointer to both the beginning and the end of a region of free memory:

```
TYPE
  FreeRec = RECORD
              OrgPtr,EndPtr : Pointer
            END;

  FreeList = ARRAY[0..8190] OF FreeRec;
```

From this declaration (which is private to the **System** unit and may not be accessed by your programs unless you declare it yourself), you can see that there is a limit of 8,191 noncontiguous blocks of free memory on the heap. That sounds like a lot of blocks, but if you're not careful with something like a text editor application that creates and disposes of text lines of various lengths as variables on the heap, you can run out in a hurry. *Turbo Pascal does not do "garbage collection."*

You can find the free list through a predefined pointer called **FreePtr**. The segment part of **FreePtr** is set to the top of heap memory minus $1000. The offset portion of **FreeList** points to the first available free block descriptor record (type **FreeRec**). When the offset is 0, the free list is empty.

That's how memory is laid out within a Turbo Pascal program. The next several sections will explain ways to examine and change locations in memory.

23.2: ABSOLUTE VARIABLES

Many computers locate certain functions at certain specific addresses in memory. The IBM PC's Color Graphics Adapter, for example, locates its screen buffer at segment $B800. If a program wishes to read from or write to a graphics buffer, it needs to take the absolute memory location of the buffer into account.

Turbo Pascal allows you to declare variables at specific places in memory. These are called *absolute variables.* They are declared as ordinary variables are, with the addition of the reserved word **ABSOLUTE** followed by the machine address of the first byte of the variable:

```
TYPE
  CGABuff = ARRAY[0..16383] OF Byte;

VAR
  GraphBuff : CGABuff ABSOLUTE $0B800 : $00
```

IBM's Color Graphics Adapter board keeps its display buffer at segment $B800 offset 0, as shown here. The segment address is given first, separated from the offset with a colon.

Defining an absolute variable allocates nothing in your data space. When the program needs to read or write to the variable, it goes directly to the physical address that you gave it in the source code. *No protection exists for overwriting critical system resources.* If you give it the wrong address, the compiled code will freely write to that address, even if to do so will crash the system bloodily.

The address given in an absolute variable declaration need not be a numeric literal. Named constants are also legal:

```
CONST
  RefreshBase = $0B000 : $0;

VAR
  GraphBuff : CGABuff ABSOLUTE RefreshBase;
```

You may give the address in one additional way: By specifying a variable name. The address of that variable becomes the address of the absolute variable. It is important to understand what this does. It maps an absolute variable *on top of* a variable declared previously. *The contents of the previously declared variable have nothing to do with where Turbo Pascal locates the absolute variable.*

An example may make this clearer:

```
VAR
  InChar    : Char;
  BuffByte : Byte ABSOLUTE InChar;
```

What we have here are two variables, each one byte in size, occupying the *same* location in memory. At compile time Turbo Pascal simply replaces the name of the variable **InChar** with **InChar**'s address in the static data area. **InChar** and **BuffByte** have incompatible types, but by placing a value in one the same value may be placed in both.

This may look familiar to you. It is, in effect, the same thing as a free union variant record (see Section 9.4) described a little more tersely. In both cases, two or more data items occupy the same physical memory, so that accessing one will access the others, regardless of data type. It is used mostly to cheat on Pascal's strong typing restrictions. The same warnings that apply to using free union variant records apply here: Know what you're doing.

Without saying much about it, we offered a pair of examples of the use of absolute variables in Section 19.10, while discussing **BlockRead** and **BlockWrite**. **GLoad** and **GSave** declare an array of 16,384 bytes at an absolute location corresponding to the IBM PC Color Graphics Adapter graphics refresh buffer. This allows the Pascal code to read and write graphics data between disk and the refresh buffer.

23.3: ABSOLUTE ADDRESS FUNCTIONS

Turbo Pascal has extended the Standard Pascal language definition with a number of functions that deal in actual physical machine addresses. Use them with care. Like most Turbo Pascal features found in this section, they can wreak havoc on your system if used incorrectly.

Addr

Knowing where an entity is in memory allows a lot of power to deal with nonstandard machine hardware. Turbo Pascal includes the **Addr** function for returning a machine address pointer to any instantiated variable, function, or procedure. **Addr** is predeclared this way:

```
FUNCTION Addr(<variable or function or procedure>) : Pointer;
```

There is an "alias" for **Addr**; that is, another identifier standing for the same function: @. In other words, these two expressions return the same pointer value:

```
BufPtr := @VideoBuffer;
BufPtr := Addr(VideoBuffer);
```

Note that @ does not require parentheses, and simply prefaces the identifier whose address it returns.

The pointer returned by **Addr** is a generic pointer that is compatible with any pointer type. Generic pointers should be considered pure machine addresses, not tied in any way to any specific data type. The following short program shows one example of **Addr** in use:

```
PROGRAM Pointers;

TYPE
  PChar = ^Char;

VAR
  CH      : Char;
  B       : Byte;
  CharPtr : PChar;

BEGIN
  CH := 'Z';
  B := 65;
  CharPtr := Addr(B);
  CH := CharPtr^;
  Writeln('CH=',CH);
END.
```

When run, it displays:

```
CH=A
```

In this example, **Addr** is used to create a pointer to a variable of type **Byte**, *and assign that pointer to a pointer-to-character pointer variable.* What this means is that **CharPtr** now points to **B**, a variable of type **Byte**, and the value stored in **B** can thus be assigned to character **CH** without running afoul of Pascal's strong type checking.

To be fair, the same effect could (and should) have been done with the **Chr(X)** transfer function. We used the **Char** and **Byte** types in the example program because they are simple and easy to understand. Tricky type conversion like this is usually done with larger and more complicated data types, typically records, arrays, or structured buffers dictated by your hardware or foreign software. *It should be obvious that you can get into trouble this way,* normally by using pointers to assign between data types that are of different physical sizes. If you assign a value in a record variable that is 200 bytes long to a record variable of a different type that is only 150 bytes long, 50 bytes worth of adjacent variables and code could be overwritten, with unpredictable, but predictably unpleasant, consequences. Do what you must, but *know what you're doing.*

Addr can only return a pointer to a unique, named entity that currently exists in memory somewhere. **Addr** cannot be used to regenerate a pointer to a "lost" dynamic variable. Nor can **Addr** reach "down into" a procedure or function from outside and return a pointer to a local variable, function, or procedure. Local entities cannot be accessed until they are instantiated; in other words, a procedure's local variables literally do not exist until that procedure is invoked. Finally, **Addr** cannot return the address of a simple constant, because there is no single location for a simple constant value. The compiler builds simple constants into the code wherever the constant is named. There are as many copies of a constant in a program as there are uses of the constant's identifier, so no single address exists for **Addr** to return.

Typed constants, on the other hand, exist at only one place in the program's data segment, and thus have a definite address that may be returned by **Addr**.

The 8086 Segment Functions

Turbo Pascal includes several built-in functions for returning important 8086 segment register values. All machine addresses in an 8086 computer have two 16-bit portions: the segment and offset. The segment portion of an address is one of 65,536 overlapping 64K regions of memory that begin every 16 bytes in the 8086 address space. The offset is a distance in bytes into one of those segments. Together, the segment and offset can specify each one of 1,048,576 separate memory locations.

Several registers within the 8086 specify which of those 65,536 segments are to be used for certain purposes. There is a *code segment,* which specifies the segment in which currently executing code must exist; a *data segment* in which data items must reside to be accessible to certain CPU manipulations, and the *stack segment* in which the CPU sets up its system stack.

There are times when you may want to examine or use the values in these registers. Turbo Pascal provides three functions to return segment register values:

CSeg returns the current value of the code segment register. This may change as different parts of the program are executed, since a Turbo Pascal program may contain more than one code segment.

DSeg returns the current value of the data segment register. As there is only one data segment in any single Turbo Pascal program, this value should not change during program execution.

SSeg returns the current value of the stack segment register. Again, there is only one stack segment, so this value should not change during the course of program execution.

All of these functions return a value of type **Word**.

Seg and Ofs

These two functions are used to return the segment and offset of a data item, procedure, or function:

```
FUNCTION Seg(<data item/procedure/function>) : Word

FUNCTION Ofs(<data item/procedure/function>) : Word
```

<Data item> may be one element of an array, or a field in the middle of a record. If applied to an array or a record as a whole, **Seg** and **Ofs** return address components of the first byte of the named data item. **<Data item>** may not be a file.

Ptr

The **Ptr** function takes a segment and offset value and combines them into a generic pointer that may be assigned to any pointer type. **Ptr** is useful for generating a pointer to machine resources that exist at a fixed location in its memory space.

```
FUNCTION Ptr(Segment,Offset : Word) : Pointer
```

The segment and offset must be expressed as two word values separated by a comma.

Peeking and Poking with the Mem Array

Actually, using the **Ptr** function to access an absolute location in machine memory is more work than you need to go through to simply do that. Turbo Pascal gives you direct read/write access to *all* of your computer's memory space by way of the **Mem**, **MemW**, and **MemL** arrays.

The elements of the **Mem** array are of type **Byte**. As mentioned above, machine addresses on 8086 systems have two parts, a segment and an offset. Both parts of the address must be given, separated by a colon:

```
HardwareFlag := Mem[$0030:$0410];
```

This is another way of allowing you access to the IBM PC's hardware configuration flag byte at $0030:$0410. It is somewhat terser than using the **Ptr** function to build a pointer to it, but has no other advantages.

Similar to **Mem** is **MemW**, an array whose elements are of type **Word** rather than type **Byte**. Otherwise it is identical to **Mem** in terms of operation. Both the segment and offset part of the address must be present in the index, separated by a colon:

```
DeviceCount := MemW[$F000:$34A6]
```

Finally, the **MemL** array's elements are of type **LongInt**; that is, signed 32-bit integers. Oddly, there is no unsigned 32-bit integer type in Turbo Pascal. Otherwise, it operates identically to **Mem** and **MemW**. An array of pointers somewhere in memory, such as the 8086 interrupt vector jump table starting at $0:$0, can be accessed via **MemL**, since long integers and pointers are the same size. Just remember to type cast the long integer returned by **MemL** onto a pointer:

```
PrtSc := Pointer(MemL[0:20]);
```

(See Section 12.3 for more about casting one type onto another.)

Using Untyped VAR Parameters

This is a feature of Turbo Pascal that is completely unique as far as I know. It will probably seem extremely peculiar to you. I have included it in this section because it is, in a sense, yet another absolute address function built into the way Turbo passes parameters to procedures and functions.

You might remember from Section 14.1 that all subprogram parameters must have a type. In Turbo Pascal this is only half true; **VAR** parameters do *not* have to have a type. This, for example, is a completely legitimate procedure header:

```
PROCEDURE VarDump(VAR Target; ItSize : Integer);
```

Any variable at all, regardless of its size or type, may be passed to **VarDump** in its parameter **Target**. This is what untyped **VAR** parameters are for: To allow you to sneak around Pascal's strict type checking and pass any variable to a function or procedure in cases where the type of the variable is not important.

To explore the concept, let's build a debugging tool. Procedure **VarDump** is a handy thing to have in your toolkit: It will accept any variable in **Target** and give a hex dump of the contents of that variable on your screen or printer. Inserting invocations of **VarDump** into your program will allow you to "peek" at the contents of critical variables while your program runs. If you have Turbo Pascal 5.0 you don't need this; a watch on a variable is infinitely preferable (see Section 30.4).

You may have already inserted **Writeln** statements in your programs to display the values of printable data items like integers, reals, and characters. **VarDump** will show you what's in data items incompatible with **Writeln** (sets, records, etc.) or which contain unprintable binary data.

```
{->>>>VarDump<<<<--------------------------------------------}
{                                                            }
{ Filename : VARDUMP.SRC -- Last Modified 7/14/88            }
{                                                            }
{ This routine will display a hex dump of any variable or      }
{ typed constant passed in untyped VAR parameter Target.       }
{ VarDump calls WriteHex; be sure WriteHex is available.       }
{                                                              }
{                                                              }
{                                                              }
{--------------------------------------------------------------}

PROCEDURE VarDump(VAR Device : Text; VAR Target; ItSize : Integer);

CONST
  Printables : SET OF Char = [' '..'}'];

VAR
  I,J       : Integer;
  Full,Left : Integer;
  DumpIt    : ARRAY[0..MAXINT] OF Byte ABSOLUTE Target;

PROCEDURE DumpLine(Offset,ByteCount : Integer);

VAR
  I : Integer;

BEGIN
  FOR I := 0 TO ByteCount-1 DO                { Hex dump the data }
    BEGIN
      WriteHex(Device,DumpIt[(Offset*16)+I]);
      Write(Device,' ')
    END;
  FOR I := 0 TO 56 - (ByteCount*3) DO Write(Device,' ');  { Space interval }
    Write(Device,'|');                        { Show first boundary bar }
  FOR I := 0 TO ByteCount-1 DO                { Show printable equivalents }
    IF Chr(DumpIt[(Offset*16)+I]) IN Printables THEN
      Write(Device,Chr(DumpIt[(Offset*16)+I]))
    ELSE Write(Device,'.');
  Writeln(Device,'|')                         { Final boundary bar }
END;


BEGIN
  Full := ItSize DIV 16;    { Number of 16-byte chunks in Target }
  Left := ItSize MOD 16;    { 'Leftover' bytes after last 16-byte chunk }
  FOR I := 0 TO Full-1 DO   { Not executed if less than 16 bytes in Target }
    DumpLine(I,16);
  IF Left > 0 THEN          { Not executed if size of Target divides by 16 }
    DumpLine(Full,Left);
  Writeln(Device)           { Space down one line after dump }
END;  { VARDUMP }
```

Before you look too closely at the actual code making up **VarDump**, look at its parameter line and variable declarations. **Target** has no type. The identifier **Target** is followed immediately by a semicolon.

If it has no type, what can be done with it? Not many things; it is incompatible with *all* other data types. You cannot assign either to **Target** or from **Target**. Neither can **Target** take part in any sort of expression. Without a data type, Pascal's runtime code cannot determine what sort of data exists in **Target** nor even how large it is. About all the information an untyped **VAR** parameter carries with it into a procedure is the address at which the actual variable being passed to the parameter begins.

That, however, can be a lot.

The declaration of the local variable **DumpIt** is the key:

```
DumpIt : ARRAY[0..MaxInt] OF Byte ABSOLUTE Target;
```

In Turbo Pascal, a variable declared as **ABSOLUTE** exists at a specific location in machine memory given by the address after the keyword **ABSOLUTE**. In this case the address is not an address but **Target**, the untyped **VAR** parameter.

Target provides the absolute starting address for **DumpIt**, as large an array of bytes as Turbo Pascal can declare. **DumpIt** begins at the same address as **Target**, and since **DumpIt** is *very* large, it will most likely be larger than any variable you pass in **Target**. By addressing bytes in **DumpIt** you can access any data that exists in the variable passed in **Target**, regardless of what data type that variable happens to be. **DumpIt** is *mapped* onto whatever variable is passed in **Target**. We've already seen how one variable may be mapped onto another via **ABSOLUTE**. Here, a *parameter* is mapped onto a variable, allowing any parameter to become a simple array of bytes.

You might worry a little about mapping a 32,767 byte variable over an actual parameter that may only be two or three bytes long. Nothing will happen to data or code beyond the end of your actual parameter *if* you know the actual parameter's size and avoid disturbing bytes beyond that size boundary. This is what the **ItSize** parameter does. Since nothing in **Target** tells the code how large its actual parameter is, **ItSize** must be passed separately. Of course, if you pass a value larger than the value of the actual parameter and then alter the actual parameter, you run the risk of disturbing other code and data, with unknown and probably unpleasant effects.

The **Device** parameter allows you to send the output of **VarDump** to your console, printer, or a text file. Pass the name of an open text file in **Device** and output will be written to that text file.

Any time you need to pass several different data types to a procedure in a single parameter, this is the way to do it. This is *not* an ISO Standard Pascal feature, nor does it exist (to my knowledge) in any other implementation of Pascal. More examples of its use will arise in the bit manipulation procedures described in Section 23.5.

23.4: HIGH-SPEED BLOCK MOVES AND FILLS

There are certain operations that are very easy and fast in assembly code, and very ponderous and slow when done in Pascal. Moving large blocks of data from one place to another can take a lot of time, often when time is critical, such as when swapping entire CRT screen buffers in and out of display RAM. Turbo Pascal provides a number of built-in routines to do this sort of work nearly as quickly as it could be done in pure assembly language.

Move

Move provides a very fast, general purpose block move function for use on a byte level. **Move** is predeclared this way:

```
PROCEDURE Move(<source>,<destination>,Count : Integer);
```

Both **<source>** and **<destination>** may be any type at all, and need not be the same type. This leniency allows you to move data wholesale from records into screen buffers, etc. **Count** is the number of bytes of data you wish to move.

Ordinarily, data is moved from the first, leftmost byte of **<source>** to the first, leftmost byte of **<destination>**. The exception is when **<source>** and **<destination>** overlap (see below). **<Destination>** may also be an element in an array, allowing you to move data into the middle of an array rather than just to the beginning of an array. **Count** bytes are moved. If we had this situation:

```
VAR
  MyBuff, YourBuff : ARRAY[1.255] OF Byte;

Move(MyBuff,YourBuff,128);
```

MyBuff[1] would first be moved to **YourBuff[1]**. Then **MyBuff[2]** would be moved to **YourBuff[2]**, and so on. This works *very* quickly compared to using a **FOR** loop to do the move. Also, a **FOR** loop would not be able to move data between different data types.

The most obvious application of **Move** is in tossing screen and other buffers around. One application of this is in the creation of instantaneous help displays that appear instantly on the press of a function key. The trick is to save the current contents of the screen out to a buffer on the heap. This can be done easily using **Move**. With the screen safely tucked away, the display can be cleared, and help information shown to the user. Then, once the user presses a key to indicate that he's done with the help display, **Move** works in the opposite direction, and brings the saved screen back from the heap and loads it into the screen buffer. Nothing was changed, and it happens fast.

Such a help system is given in the procedure **ShowHelp** on the next page. **ShowHelp** needs only the addition of your own help information, and you can use it in your own programs. Simply create a procedure **ShowHelpData** that displays whatever you like on the screen. The help information shown here is for the **JTerm** terminal program given in Section 23.7.

Note the two typed constants, **ScreenX** and **ScreenY**. These indicate the size of the screen in which the routine is used, and allow **ShowHelp** to calculate how much data to move with **Move**. If you use the 43-line mode on the EGA, or the 50-line mode in the VGA, or some other non-standard screen mode like the 66-line mode using the Genius VHR display, change **ScreenX** and **ScreenY** to correspond to your screen size. Because they are typed constants, they can be modified at runtime if your program has the ability to change screen sizes on command from the user.

```
{->>>>ShowHelp<<<<-------------------------------------------}
{                                                           }
{ Filename : SHOWHELP.SRC -- Last Modified 7/14/88          }
{                                                           }
{ This is a simple, single-screen help system that will save  }
{ the underlying screen, show a single screen's worth of help }
{ information, and then restore the underlying screen once any }
{ key is pressed.  You must customize the ShowHelpData routine }
{ with your own help information; the data shown here is for  }
{ the JTERM terminal program from Section 23.7.             }
{                                                           }
{                                                           }
{                                                           }
{-----------------------------------------------------------}

PROCEDURE ShowHelp;    { Shows a help screen display on press of F1 }

CONST
  ScreenX = 80;
  ScreenY = 25;   { This could be 43 for the EGA; 50 for VGA; 66 for Genius }

VAR
  XSave,YSave : Integer;
  VidSegment : Word;
  VideoBufferSize : Word;
  SavePtr   : ^Word;
  VideoPtr  : ^Word;
  VideoSeg  : Word;
  Dummy     : Char;


FUNCTION Monochrome : Boolean;

VAR
  Regs : Registers;

BEGIN
  INTR(17,Regs);
  IF (Regs.AX AND $0030) = $30 THEN Monochrome := True
    ELSE Monochrome := False
END;


PROCEDURE SaveScreenOut;

BEGIN
  XSave := WhereX; YSave := WhereY;        { Save the underlying cursor pos. }
  VideoBufferSize := ScreenX*ScreenY*2;    { E.g., 25 X 80 X 2 = 4000 bytes  }
  { Allocate memory for stored screen: }
  GetMem(SavePtr,VideoBufferSize);
  IF Monochrome THEN VidSegment := $B000 ELSE  { Get a screen buffer origin }
    VidSegment := $B800;
  VideoPtr := Ptr(VidSegment,0);           { Create a pointer to the buffer  }
  Move(VideoPtr^,SavePtr^,VideoBufferSize);  { Save screen out to the heap   }
END;


PROCEDURE ShowHelpData;
```

```

BEGIN
  GotoXY(30,3); Writeln('>>>JTERM<<<');
  Write('          From COMPLETE TURBO PASCAL 5.0, by Jeff Duntemann');
  GotoXY(1,7);
  Writeln('The default communications parameters are 1200, 8, N, 1');
  Writeln('You can use JTERM at 300 baud by invoking it this way:');
  Writeln;
  Writeln('  C:\>JTERM 300');
  Writeln;
  Writeln('Currently active commands are:');
  Writeln;
  Writeln('  F1:     Displays this help screen');
  Writeln('  Ctrl-Z: Clears the screen');
  Writeln('  Ctrl-X: Hangs up and exits JTERM');
END;


PROCEDURE BringScreenBack;

BEGIN
  Move(SavePtr^,VideoPtr^,VideoBufferSize);  { Bring screen back from heap  }
  FreeMem(SavePtr,VideoBufferSize);          { Free up the meap memory      }
  GotoXY(XSave,YSave);                      { Put the cursor back where it was }
END;


BEGIN  { ShowHelp }
  SaveScreenOut;
  ClrScr;
  ShowHelpData;
  GotoXY(20,23); Write('Press any key to continue...');
  REPEAT UNTIL KeyPressed;
  Dummy := ReadKey;
  IF Dummy = Chr(0) THEN Dummy := ReadKey;
  BringScreenBack;
END;
```

Move can also be used to move data from one part of a data structure to another area within the *same* data structure. The source and destination areas may overlap; the **Move** procedure is just smart enough to determine how to perform the move so that data does not overwrite itself during the move.

Move is a low-level, byte-oriented procedure. It is very fast, but rather devil-may-care; it does not recognize data types nor the boundaries of variables. **Move** will obediently move a 200-byte variable into a 100-byte variable. However, in the process you will overwrite 100 bytes of adjacent storage, with possibly disastrous consequences. One good idea is to use **SizeOf** (see below) to supply the **Count** parameter when moving one variable into another. That way, even if you change the physical size of the variable to be moved, the **Count** figure will always automatically be exactly right.

FillChar

Turbo Pascal provides a way to fill an area of memory with some byte value. **FillChar** should be considered a high speed buffer-blanker. It is predeclared this way:

```
PROCEDURE FillChar(<destination>; Count : Word; <ordinal>);
```

As with **Move**, **<destination>** may be any data type. **Count** indicates how many bytes are to be filled. The last parameter may be any ordinal variable, constant, or expression. It provides the value with which **FillChar** will fill **<destination>** from its beginning to **Count** bytes past its beginning. It may be any type at all, *as long as the representation of that type is only one byte in size.* If **<destination>** is an array, you may also specify an element of the array as a starting point, to enable you to begin filling in the middle of **<destination>** if you need to:

```
VAR
  TBuff : ScreenBuff;

FillChar(TBuff,SizeOf(TBuff),0);
FillChar(TBuff[4096],2048,7);
```

As with the block move procedures, **FillChar** does not respect boundaries of variables or code segments. If **Count** takes the fill beyond the edge of **<destination>**, whatever lies beyond will be obliterated by the fill. Use **FillChar** with care.

SizeOf

The **SizeOf** function allows your program code to "know" how large a variable or data type is. Within the bounds of ISO Pascal there is no need for this; but certain Turbo Pascal extensions need to know the size, in bytes, of the creatures they act upon. **SizeOf** is predeclared this way:

```
FUNCTION SizeOf(<data type or variable>) : Integer;
```

The parameter may only be a data type or a variable; **SizeOf** cannot determine the sizes of functions, procedures, or constants.

The **FillChar** procedure (see above) is the simplest example of this. **FillChar** fills a variable with a given byte-size value. It needs to know how large the variable is to do this:

```
FillChar(MyRec,SizeOf(MyRec),0);
```

This statement takes the record **MyRec** and fills it with binary 0. The middle parameter tells **FillChar** how large a region of memory must be filled with 0s.

You could, of course, simply fill in the size parameter with a literal constant:

```
FillChar(MyRec,144,0)
```

Assuming that **MyRec** is a record type that is 144 bytes in size, this will work handily. However, if you ever change the definition of **MyRec**'s record type and thus its size, you could be in serious trouble. **FillChar** does not recognize the boundaries of data items; it starts at the beginning of the destination variable and lays down as many filler bytes as the size parameter calls for. If you pare down **MyRec** to occupy only 120 bytes, **FillChar** will still lay down 144 binary 0's, overwriting adjacent code or data.

Using **SizeOf** with **FillChar** ensures that **FillChar** will always *exactly* fill the destination variable, neither a byte more nor a byte less, regardless of how often you change the definition of the destination variable's type.

With care, of course, you don't need **SizeOf**. But you might as well allow the compiler to take as much work and worry off your hands as it can. Bugs proceeding from hard-coded size parameters are insidious, inconsistant, and completely avoidable. Use **SizeOf** with all the move and fill built-ins: **FillChar**, **Move**, and also with **BlockRead** and **BlockWrite**.

23.5: BIT MANIPULATION

Dealing with numbers and characters has been the traditional task of Pascal. Dealing with the computer itself, at a very low level, or dealing with machinery existing in the outside world demands more precision than the ability to grasp 8 or 16 bits at a time. In systems programming, where memory is usually at a premium, important information is often stored in a single bit. Often, eight information bits are placed together in a byte. This is often called a *flag byte*, because it contains eight one-bit flags. Though the computer reads and writes the flag byte as a unit, it must be able to set, clear, and individually interpret each 1-bit flag in the byte.

Flag bytes like this are frequently used to control peripheral chips like serial port controllers, parallel port controllers, and baud rate generators. These chips are generally accessed as machine I/O ports. Because of the way our computers work, speaking of reading I/O ports and working with bits is best done together.

Setting, Clearing, and Testing Bits

Turbo Pascal has no built-in facilities for testing, setting, or clearing bits. To manipulate bits you must deal with whole bytes or integers and use Turbo Pascal's bitwise logical operators to *mask* out the bits you wish to leave alone. We looked at these bitwise operators in Section 11.4. Do refer back to that section if their operation is unclear to you.

Working with One Bit at a Time

In situations where you need to isolate one single bit from an 8- or 16-bit variable, it's quite easy to create a set of bit manipulation functions that will work on any such variable, regardless of type. In this section we'll show you how such functions work.

TestBit

First of the three is a function that tests whether a particular bit is 1 (*set*) or 0 (cleared). **TestBit** returns **True** if the selected bit is set and false if it is cleared.

The **BitNum** parameter specifies the bit that you wish to test. The bits within an 8-bit variable are numbered 0 through 7. In a 16-bit variable they run from 0 through 15. Bit 0 is the least significant bit; in the pure binary representation of a number it represents 2 to the 0th power, or 1. Bit 1 represents 2 to the first power, or 2; and so on.

The following demonstration program contains the **TestBit** function. The program prompts you for an integer, and then prints the binary equivalent of the integer you enter. It does this by testing each bit of the integer and printing a 1 if the bit is set and 0 if the bit is cleared. To exit the program enter 0:

```
{--------------------------------------------------------------}
{                          BINARY                              }
{                                                              }
{                 Bit test demonstration program               }
{                                                              }
{                          by Jeff Duntemann                   }
{                          Turbo Pascal V5.0                   }
{                          Last update 5/22/88                 }
{                                                              }
{                                                              }
{                                                              }
{--------------------------------------------------------------}

PROGRAM BinaryDemo;

VAR
  I,J : Integer;


FUNCTION TestBit(VAR Target; BitNum : Integer) : Boolean;

VAR
  Subject : Integer ABSOLUTE Target;
  Dummy   : Integer;

BEGIN
  Dummy := Subject;
  Dummy := Dummy SHR BitNum;
  IF Odd(Dummy) THEN TestBit := True
    ELSE TestBit := False
END;
```

```


BEGIN
  REPEAT
    Write('>>Enter an integer (0 to exit): ');
    Readln(I);
    FOR J := 15 DOWNTO 0 DO
      IF TestBit(I,J) THEN Write('1') ELSE Write('0');
    Writeln; Writeln
  UNTIL I = 0
END.
```

TestBit tests bit **#BitNum** in **Target**. If the bit is set, the function returns **True**. If the bit is cleared, the function returns **False**. **Target** is an untyped VAR parameter, as we described in the previous section. Using an untyped **VAR** parameter allows us to pass any 8- or 16-bit variable to **TestBit** regardless of its type. A larger variable will in fact be accepted by **TestBit** in its **Target** parameter; however, **TestBit** will ignore any bytes after the first 2.

BinaryDemo produces output like this:

```
>>Enter an integer (0 to quit): 17367
0100001111010111

>>Enter an integer (0 to quit): 16
0000000000010000

>>Enter an integer (0 to quit): 0
0000000000000000
```

How does this work? You only want to examine 1 bit out of the 8 or 16 in **Target**. There is already a standard Pascal procedure that tests a single bit in an 8-or 16-bit variable, although it may not be the bit you want: **Odd(X)** tests bit 0 of X. If that bit is set, **Odd** returns a Boolean value of **True**. If the bit is cleared, **Odd** returns **False**.

Odd works this way because all odd numbers, expressed as binary bit patterns, will have bit 0 set to 1. To test a bit other than bit 0 you must move your desired bit down into the 0 position. This is done with the **SHR** (SHift Right) operator. **SHR** shifts the desired bit from its position down to bit 0. **Odd** then tests to see if that bit is set or cleared.

SetBit

Turning on one single bit of an 8- or 16-bit variable is the job of **SetBit**:

```
PROCEDURE SetBit(VAR Target; BitNum : Integer);

VAR
```

```
  Subject : Integer ABSOLUTE Target;
  Mask    : Integer;

BEGIN
  Mask := 1 SHL BitNum;
  Subject := Subject OR Mask
END;
```

SetBit sets bit **BitNum** in **Target**. How? This time we're setting up an actual bitmask. **Mask** starts out with a single set bit in bit 0, the least significant and rightmost bit in an integer. If we wanted to set bit 0, this would be sufficient. To set the other bits, we have to move that single set bit "over" to the bit position we wish to set. This is done by shifting **Mask**'s single set bit leftward with Turbo Pascal's **SHL** operator. The number of bits it must be shifted is the same number of bits as the bit number we wish to set. For an example, say we wish to set bit 6 of a variable. Here's **Mask** before and after the shift:

```
Mask containing the integer value 1:

          0000000000000001

Mask after shifting 6 bits leftward:

          0000000000100000
```

Now we have a proper bitmask. To use it, apply the mask against the target variable (say, a value of 129) with the bitwise logical operator **OR**:

```
0000000010000001
      OR
0000000001000000
    yeilds
0000000011000001
```

Follow each bit column down, applying the **OR** operator between each bit in the value (top) with the mask (middle) to produce the result (bottom.) If bit 6 of the value was cleared, the OR with bit 6 in the mask will set it. If bit 6 in the value was already set, the operation will have no effect.

ClearBit

The flipside of setting bits is clearing them, and we can define a procedure **ClearBit** to do that job:

```
PROCEDURE ClearBit(VAR Target; BitNum : Integer);

VAR
  Subject : Integer ABSOLUTE Target;
  Mask    : Integer;

BEGIN
  Mask := NOT(1 SHL BitNum);
  Subject := Subject AND Mask
END;
```

ClearBit clears bit **BitNum** in **Target**. **Target**, as with **TestBit** and **SetBit**, may be any 8- or 16-bit variable (types **Char**, **Byte**, **Integer**, or **Boolean**). **Target** may be larger than 2 bytes, but any bytes after the second will be ignored. Remember that setting a bit puts that bit to a binary 1; and clearing a bit puts that bit to a binary 0. The mechanism here is almost identical to that of **SetBit**, save that we have inverted the mask and are using the **AND** bitwise operator rather than **OR**:

```
0000000011000001
      AND
1111111110111111
    yeilds
0000000010000001
```

Again, follow each bit column down, applying the top bit (the value) upon the middle bit (the mask) to yield the result on the bottom. If you are still a little fuzzy on how bitwise **AND** and **OR** work, refer back to Section 11.4.

Working with Several Bits at a Time

The three procedures just given work only with one bit at a time. The general method of creating a mask and using the bitwise logical operators works as well for several bits as it does for one, since each bit is operated upon separately.

A good example is the **WriteHex** procedure we have used in the **HexDump** program and **VarDump** procedure described previously:

```
{<<<< Writehex >>>>}


{ Described in section 23.5   --   Last mod 12/5/87  }



PROCEDURE WriteHex(VAR Device : Text; BT : Byte);

CONST
```

```
  HexDigits : ARRAY[0..15] OF Char = '0123456789ABCDEF';

VAR
  BZ : Byte;

BEGIN
  BZ := BT AND $0F;
  BT := BT SHR 4;
  Write(Device,HexDigits[BT],HexDigits[BZ])
END;
```

WriteHex prints a byte in hexadecimal notation. In hexadecimal notation, each 4-bit half of a byte (a *nybble*) has its own base-16 digit from 0 to F. The first line of code in **WriteHex** isolates the low nybble of the byte by masking out the high nybble. If the byte happened to be 122 (hex 7A) the mask operation would look like this:

```
01111010   ($7A)
  AND
00001111   ($0F)
 yeilds
00001010   ($0A)
```

The **AND** operation forces all the 1 bits in the high nybble to 0 bits, while retaining the low nybble without change.

Isolating the high nybble is done a slightly different way. We need the high nybble, but if we simply mask out the low nybble we will have not $7 but $70, which is far outside the range of the array **HexDigits**. We must change $70 to $07. It's done by shifting the $7A value four bits to the right. The low nybble gets "bumped off the edge" into oblivion, and 0 bits are fed into the byte from the high side to take their place:

```
Before shifting:    01111010   ($7A)

After shifting:     00000111   ($07)
```

Not only does this bring the high nybble down to the low nybble where we need it; it zeroes out the high nybble to keep it out of mischief.

BT and **BZ** now contain $07 and $0A respectively. By indexing into **HexDigits** with those values ($0A is equivalent to decimal 10, recall) we come up with the characters 7 and A. Voila! Hex on the half byte!

If the working of **SHL** and **SHR** still puzzles you, review their introduction back in Section 11.2.

23.6: DEALING WITH I/O PORTS

Most PC peripheral devices are *I/O mapped;* that is, they are reached through special I/O instruction opcodes rather than mapped onto RAM memory. An "I/O port" is actually a code number that is sent out onto the address bus, along with some sort of signal (usually a special pin on the CPU chip itself) indicating that an I/O operation is underway. If a physical device exists on the address bus that responds to that code number, data will be sent or received between the CPU and the peripheral device.

Turbo Pascal accesses I/O ports through a pair of predeclared arrays: **Port** and **PortW**. **Port** sends or receives one byte at a time. **PortW** sends or receives a word quantity, which contains 2 bytes. The range of legal indices for **Port** and **PortW** is 0 through 65535. **Port** and **PortW** may be considered to be predeclared arrays as follows:

```
Port : ARRAY[0..65535] OF Byte;

PortW : ARRAY[0..65535] OF Word;
```

Port

When you read a byte from the **Port** array, you are actually reading a byte from the port whose number is the subscript of the element read:

```
StatByte := Port[4];
```

This statement reads from I/O port 4 by reading element 4 of the array **Port**.

In a similar fashion, writing a byte to an element of **Port** actually writes a byte to the I/O port whose number is the subscript of the element written to:

```
Port[$10] := SIOSetup;
```

Here the byte **SIOSetup** is written to I/O port 16 (hex 10) by assigning the byte to element 16 of **Port**.

As far as your code is concerned, **Port** is treated no differently from an array of **Byte** with two exceptions:

1. You may not reference the entire array as the identifier **Port** without a subscript. Whenever **Port** is used it *must* have a subscript!
2. Individual elements of **Port** may not be passed to procedures in reference (**VAR**) parameters.

Port may be indexed by constants, variables, or expressions.

PortW

PortW is, in essence, a 16-bit version of **Port**. Its index type is **Word** and its elements are also type **Word**. **PortW** should be used *only* when you must transfer 16 bits at a time between your program and an I/O port. We're used to calling the IBM PC a 16-bit machine, but that's questionable, since the 8088 CPU chip presents a very definite 8-bit bus to the outside world. Attempting to read 16 bits from a single I/O port on an 8-bit bus may actually do a number of different things, depending on the electrical nature of the device that responds to the requested I/O port number. You may get garbage in the high byte of the returned integer. You may get a duplication of the low byte in the high byte. You may also read the next higher I/O port number (if one has been implemented in the system) into the high byte. Whatever happens is not likely to be meaningful for all 16 bits.

The same cautions apply when attempting to *write* 16 bits to an I/O port in the PC or XT using **PortW**. Use **Port** instead, for the PC and XT. **PortW** should work correctly on the PC/AT and its 80286-based compatibles, since the AT is a true 16-bit machine with a 16-bit bus, and also on most machines incorporating an 8086 (rather than 8088) CPU. The same holds true of machines based on the 80386 CPU. The 386 is also capable of transferring 32 bits between ports and memory, but there is no high level construct in Turbo Pascal to support this.

The following statement will fire the one-shots in the IBM PC Game Controller Adapter:

```
Port[$201] := 0; { Anything out to port $201 fires oneshots }
```

For additional examples of port I/O, see the **JTerm** program in the next section.

23.7: INTERRUPT PROCEDURES

Starting with Turbo Pascal 4.0, you can write your own interrupt service routines (ISR's) directly in Turbo Pascal, without having to resort to either assembly language or, more usually, **INLINE** machine code. **INTERRUPT** is a qualifier for procedures (but *not* functions) that causes the compiler to generate the procedure's code in the slightly special way that interrupt service routines require. A procedure header declaration for an interrupt procedure can take two forms: With register parameters, or without register parameters:

```
PROCEDURE ASimpleISR; INTERRUPT;

PROCEDURE AComplexISR(Flags,CS,IP,AX,BX,CX,DX,SI,DI,DS,ES,BP : Word);
INTERRUPT;
```

The **INTERRUPT** reserved word must be present in either case, separated from the declaration by a semicolon.

The Nature of an Interrupt Procedure

I've covered interrupts and the interrupt vector table in Section 20.2. You might refer back to that section if your understanding of the nature of interrupts is fuzzy.

In brief, the 8086 family of CPUs keeps a table of addresses in low memory. Each address points to an interrupt service routine. The first address, the one starting at 0:0, belongs to interrupt 0. The next address, which starts at memory location 0:3, belongs to interrupt 1, and so on. Software interrupts are called by the execution of an **INT <n>** opcode, where **<n>** is the number of an interrupt from 0 through 255. Hardware interrupts are generated by inputs to a chip called a Programmable Interrupt Controller (PIC) which, at least for the older PC-type machines, is an 8259.

Setting up an ISR is mostly a matter of getting the ISR's address into the correct slot in the interrupt vector table. This is easily done with the **SetIntVec** procedure from the DOS unit (see Section 20.2). That way, when an interrupt occurs, the CPU transfers control to the ISR through the address in the interrupt vector table.

When Turbo Pascal generates the code for an interrupt procedure, it generates the following assembly language *prolog:*

```
PUSH AX
PUSH BX
PUSH CX
PUSH DX
PUSH SI
PUSH DI
PUSH DS
PUSH ES
PUSH BP
MOV  BP,SP
SUB  SP, <size of local data>   ;Only if locals are present!
MOV  AX, <data segment address>
MOV  DS,AX
```

As you can see, the prolog pushes all of the CPU registers except for SP and the flags onto the stack, and saves the value of SP in the BP register. The SUB opcode will be present in the prolog *only* if your interrupt procedure has local variables. Its purpose is to make room on the stack for the local variables. Finally, Turbo Pascal's data segment address is moved into DS by way of the AX register, since segment registers cannot be loaded directly from memory.

Having Turbo Pascal's DS value allows your interrupt procedure to access global variables freely, even if the procedure is called as part of a hardware interrupt, which may occur from anywhere in the system (e.g., from within DOS or from within a BIOS routine) when the data segment may be completely unknown.

After your ISR's code is generated by the compiler, an epilog is added that undoes what the prolog did:

```
MOV   SP,BP
POP   BP
POP   ES
POP   DS
POP   DI
POP   SI
POP   DX
POP   CX
POP   BX
POP   AX
IRET
```

Something to remember: When an ISR is given control through the interrupt vector table, interrupts are disabled, and will remain disabled until the IRET instruction executes, *unless you explicitly re-enable them.* This can be done easily by adding a 1-byte **INLINE** macro (see Section 24.5) to your program:

```
PROCEDURE EnableInterrupts;

INLINE($FB);
```

Calling **EnableInterrupts** in your ISR will enable interrupts while the ISR is executing. It's not a good idea to leave interrupts disabled for more than a handful of cycles at any time, since timer tick interrupts are generated by the PC hardware every 55 milliseconds. If a timer tick interrupt happens while interrupts are disabled, your system clock will slow down by about 1/18 second. Do this enough times, and you could seriously disrupt the PC's timekeeping.

Something else to remember: If your ISR services a *hardware* interrupt (e.g., a serial port, the keyboard, or some other hardware device) you *must* send the 8259 PIC chip a signal telling it that the interrupt has been serviced *before* leaving the ISR. This is easy enough in most cases:

```
Port[$20] := $20;
```

Since the PIC will not allow another hardware interrupt until it receives this signal (called *EOI* for *End Of Interrupt*) you should send the signal soon after entering the ISR, probably right after invoking **EnableInterrupts** as described above. Remember that the PIC is involved *only* with hardware interrupts, and you need not send it the EOI signal for software interrupt ISRs.

Register Parameters

The formal register parameters in an interrupt procedure header are not true parameters. They don't take actual parameters, and they are not allocated on the stack as are normal parameters. They exist to allow you to test register values passed from a software interrupt invocation through **Intr**, or to return register values from the ISR to the program calling it. Register parameters may be modified from within the ISR, and the modified values will be loaded back into the appropriate CPU registers just before the ISR returns control to the calling logic.

Creating a Software Interrupt Service Routine

There is nothing especially difficult about writing your own software interrupt service routine. It is much easier than writing a hardware interrupt service routine, as we'll see a little later on. The bulk of the trickiness lies only in getting the address into the correct slot in the interrupt vector table, and that's not very tricky. The program below creates a simple demonstration software interrupt service routine for one of the unused interrupts, 76. It does nothing but demonstrate that it works, and that it can be called through the **Intr** procedure, with values passing in both directions through the **Registers** record.

```
{--------------------------------------------------------------}
{                          SoftIntTest                         }
{                                                              }
{       Software interrupt service routine demonstration       }
{                                                              }
{                            by Jeff Duntemann                 }
{                            Turbo Pascal V5.0                 }
{                            Last update 7/17/88               }
{                                                              }
{                                                              }
{                                                              }
{--------------------------------------------------------------}

PROGRAM SoftIntTest;

USES DOS,CRT;


VAR
  Foo : Word;
  Regs : Registers;
  OldVector : Pointer;



PROCEDURE EnableInterrupts;

INLINE($FB);



PROCEDURE SoftISR(Flags,CS,IP,AX,BX,CX,DX,SI,DI,DS,ES,BP : Word);
INTERRUPT;

```

```
BEGIN
  EnableInterrupts;
  FOO := AX;
  BX := 17;
END;



{$F+}
PROCEDURE OurExitProc;

BEGIN
  SetIntVec(76,OldVector); { Put 76 back to whatever it was before }
END;
{$F-}



BEGIN
  ExitProc := @OurExitProc;  { Link exit proc into the chain }

  FOO := 0;
  ClrScr;

  { We use 76 here because it's an unused vector }
  GetIntVec(76,OldVector);   { Save old copy...just in case }
  SetIntVec(76,@SoftISR);    { Give vector 76 SoftISR's address }

  Regs.AX := 42;       { Pass a value to SoftISR through AX }
  Regs.BX := 0;        { Zero BX before making the call }
  Intr(76,Regs);       { Trip interrupt 76 }

  Writeln('Foo=',Foo);       { Now examine FOO and BX }
  Writeln('BX =',Regs.BX);
  Readln;
END.
```

It isn't absolutely necessary to save the old contents of an interrupt vector before you write over them with the address of your own ISR. But it's good practice to leave the machine in the same state you found it after your program returns control to DOS. This is why **SoftIntTest** stores the prior contents of interrupt 76 in a **Pointer** variable during program execution, and restores those prior contents through a **SetIntVec** call before the program terminates.

The other point is that the restoration of the old vector should be done within an exit procedure. Why? No matter how your program terminates, the exit procedure will *always* execute, regardless of whether the program ended normally, through a runtime error of some sort, or through a "trap door" **Halt** instruction. If you restore the old vector in the exit procedure, you *know* that short of a total system crash, which would require rebooting anyway, you will leave the vector table as you found it.

SoftInttest shows how the register parameters may be used to pass information to and from the interrupt procedure. The value in global variable **FOO** is altered by the ISR, which copies the contents of register AX into **FOO**. The ISR also assigns a value to register BX and sends it back to the main program in the register parameter.

What Not to Do within a Software ISR

Writing your own ISRs carelessly can get you into trouble. It's much worse with hardware ISRs, but there are some points to keep in mind while writing software ISR's.

Most of the problems, with hardware as well as software ISRs, stem from the fact that most code running under DOS, including DOS itself and parts of BIOS, are not re-entrant. Re-entrant means a routine can be executed from within itself, usually because the original *thread* through the code was interrupted by a system interrupt of some sort. What usually happens is that non re-entrant code works with static tables somewhere in memory, and the second thread through the code overwrites the unfinished work of the first thread. Not all of the Turbo Pascal runtime library is re-entrant. DOS is emphatically *not* re-entrant.

On the other hand, if your ISR is part of a nonmemory resident application, you can call DOS from within the ISR because DOS will never be in control when the ISR executes. The thing to watch out for is calling a Turbo Pascal library routine from within an ISR that might itself call that interrupt. In other words, if you write a replacement ISR for some portion of video interrupt 10H, *don't have your ISR call any library routine that might possibly call INT 10H itself.* The reason why not should be obvious: If your ISR calls a routine that calls your ISR, the two routines will become locked in what is most accurately called a "deadly embrace," and you'll be reaching for the Big Red Switch.

In most cases, as long as you install your ISR *only* for the duration of your Turbo Pascal program, and only call it from within your Turbo Pascal program, you can safely call DOS functions from within your ISR. This will *not* be true if you create an ISR that remains memory resident after returning control to DOS; such programs are those mysterious, infuriating Terminate, Stay Resident (TSR) utilities that I will not be showing you how to write in this book. They deserve a book all to themselves. If you decide to strike out on your own, wear your flak helmets. A TSR can be called from anywhere in the system (from DOS, from a hardware ISR, from anywhere) and thus re-entrancy becomes a very serious problem.

One final caution: Software interrupt 1CH (the "user" timer tick interrupt) is a special case. It is a software interrupt, but it is called *only* from within the hardware interrupt ISR for interrupt 8, the timer tick. If you work with software interrupt 1CH, treat it as though it were a hardware interrupt, since it can occur within DOS, BIOS, or anywhere else. Re-entrancy problems apply.

Writing a Hardware Interrupt Service Routine

The most dangerous interrupt game of all is the hardware interrupt, a creature that can and will bite if handled carelessly. Turbo Pascal's **INTERRUPT** procedures can service hardware interrupts, but you must pay attention to more details, and touchier ones.

Hardware interrupts are generated by the serial port, by the PC keyboard, most mouse-type peripherals, and the parallel port under certain circumstances. The PC's timer generates a hardware interrupt every 55 milliseconds. Other special-purpose peripherals like LAN boards and data acquisition boards may also generate hardware interrupts. You can write ISRs for any of these interrupts, but before getting started, *learn the hardware that generates the interrupt.* As I will demonstrate shortly, hardware devices require lots of "magic numbers" and give and take through I/O ports, and one misplaced bit can send your system into the bushes.

Structurally, a hardware interrupt ISR is little different from a software interrupt ISR. One difference is that hardware interrupts are funnelled to the CPU through a Programmable Interrupt Controller (PIC) called the 8259. Once the 8259 recognizes a hardware interrupt, it blocks all further interrupts until it is reset. So with a hardware interrupt, you *must* reset the PIC before returning, or your system timer will be disabled, and it's unclear how well your machine will function. In fact, it's a good idea to reset the PIC almost immediately, so that the timer tick interrupts can occur normally, even while your ISR is in control.

An Interrupt Driven Terminal Program for the PC Serial Port

The irreducible complexity of hardware interrupt work is best shown by a simple example. The **JTerm** program shown below is a *dumb terminal* program. It does no file uploading or downloading. Its purpose is to mediate among three devices: Your keyboard, your CRT, and your serial port, which is typically connected to a modem. Data coming from the serial port is sent to your screen; data coming from your keyboard is sent to the serial port. In concept it is simplicity itself.

Why the need for interrupts? You can sit in a loop and *poll* the serial port (SIO) chip to see if it has data for you, and poll the keyboard to see if anything was typed, taking action only when data appears from one device or the other. Such polled terminal programs are common and very easy to write. The problem is one of speed.

If too much time elapses between one polling of the SIO and the next, an entire character can come in from the modem and be lost. The SIO chip doesn't wait for you to poll it; it puts a received character up on the rack and expects you to grab it. If another character comes in before the first has been grabbed, the new character will take the place of the old, and the old will simply disappear.

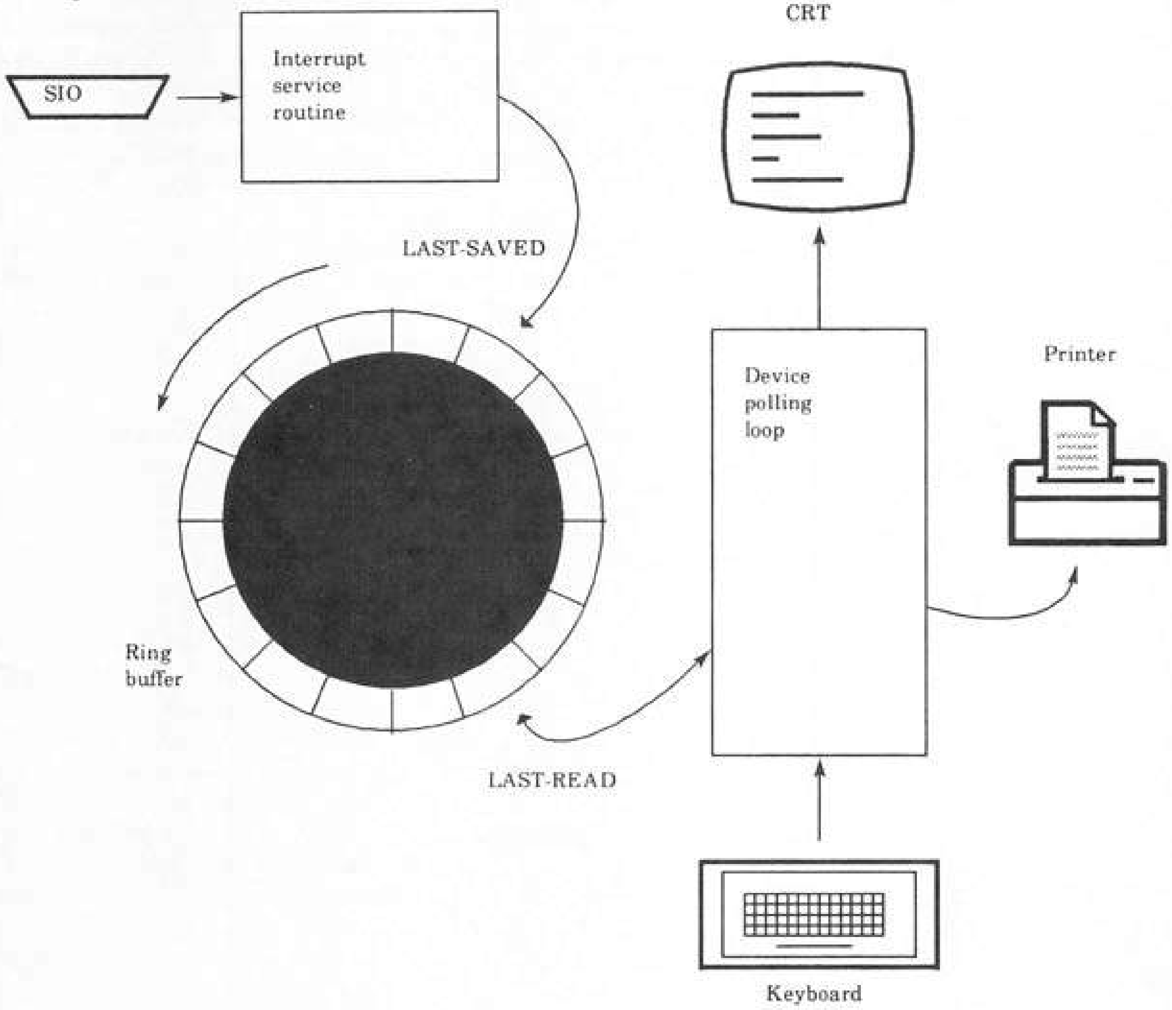
At 300 baud, characters are coming in so slowly that the computer has plenty of time between incoming characters to do what it has to do: clear its screen, move the cursor, and so on. At 1200 baud it can usually keep up, but not always. If the remote system is sending a large number of CRT control commands, the CPU can be so busy executing the commands that it will be unable to poll the SIO often enough. Lost characters are the result.

Interrupts provide the answer.

Refer to Figure 23.2 during the following discussion. This is difficult stuff. You should know how the PC's 8250 SIO chip works before attempting to fully understand **JTerm**.

Figure 23.2

A Ring Buffer for Serial Input



Central to **JTerm**'s operation is something called a *ring buffer.* Physically, a ring buffer is nothing more than an area of memory, in this case, defined as a Pascal array of 1,024 characters. Two pointers are set up to point to locations in the buffer. One, **LastSaved**, points to the last character placed in the buffer. The other, **LastRead**, points to the last character *read* from the buffer. The two pointers are incremented along the buffer as they do their jobs, as we'll describe below. What makes the buffer a ring buffer is that there is code that repoints both pointers back to the low end of the buffer once they are incremented off the high end. Metaphorically, this "wraps" the two ends of the buffer around until they meet, so that the two pointers can be incremented around and around the buffer without running off the end.

Now, how it works: The interrupt service routine **Incoming** is executed whenever the 8250 SIO chip receives a complete character from the remote system. All the routine does is increment **LastSaved** by one, and then read a character from the SIO and write it into the buffer at the location pointed to by **LastSaved**. This happens regardless of whatever else the CPU was doing at the time the interrupt occurred. If the terminal program isn't quite ready to accept another character from the SIO yet, that's OK. The character is safely stored in the ring buffer, waiting.

That's what the interrupt service routine **Incoming** does. Now look down at the main program code for **JTerm**. It's a relatively simple polling loop. When simplified, the logic works like this:

```
REPEAT
  If an incoming character is waiting,
    go read it from the ring buffer and display it.
  If a character was typed at the keyboard,
    execute it if it is a command,
    else send it to the SIO.
UNTIL we decide to quit.
```

This, in essence, is all **JTerm** does. Let's look at the code more closely:

How does **JTerm** know if a character is waiting in the ring buffer? The function **InStat** tests to see if **LastSaved** is equal to **LastRead**. If the last character saved was the last character read (think about that), nothing is waiting to be read, and **InStat** returns **False**. If the two pointers are *not* equal, it means that **LastSaved** has gotten *ahead* of **LastRead** in the ring buffer, and more characters have been saved than have been read. The terminal loop thus reads another character and displays it to the screen. **LastRead** is incremented by **InChar**, the function that actually reads characters from the ring buffer. The next time through the loop, **InStat** will again see if **LastSaved** is ahead of **LastRead**, and if it is, another character will be read until the two pointers are equal. *Whenever **LastSaved** and **LastRead** are equal, the ring buffer is considered empty.*

If the terminal loop is busy because it is executing keyboard commands, **LastSaved** can get considerably ahead (1,024 bytes ahead, in fact) of **LastRead**. That's all right, because in the long run the terminal loop will get its work done and have plenty of time to grab and display incoming characters from the ring buffer.

One of the most critical routines in **JTerm** is **SetupSerialPort**. One thing that must be done just so is the replacement of the existing serial port interrupt vector with a new vector pointing to **Incoming**. This is done with the **GetIntVec** and **SetIntVec** routines from the **DOS** unit. You must first save the old vector in a global **Pointer** variable. Then before doing anything else, link **JTerm**'s exit procedure into the exit procedure chain. Why? If *anything* crashes the program at this point, you want the exit procedure to restore the original interrupt vector. So follow this general rule: As soon as you modify an existing vector, put the machinery in place for restoring it. Once the exit procedure is safely in the chain, establish a new vector pointing to **Incoming** using **SetIntVec**.

I won't attempt to explain the rest of **SetupSerialPort** in detail, because it deals with "magic numbers" and bit masks specific to the 8250 SIO chip that are too involved to cover here. Before trying to modify any of the code in **JTerm**, you should study the INS8250 technical documentation in the *IBM Technical Reference.* Note that **JTerm** is hard-coded to support the COM1: serial port only; changing it to COM2: will involve altering several of the magic numbers.

Sending a character to the serial port, through the **OutChar** procedure, is as simple as placing the character in the serial port's Transmit Holding Buffer (THB) where it will be held until the chip has a spare moment to send it to the modem. If you're only working from the keyboard, *outbound* characters can be sent without any kind of output buffering. Once you begin sending files, however, it's a good idea to set up an output ring buffer. This can be done because the 8250 can be initialized to trigger a hardware interrupt when it is ready to send another character. Again, this is subtle, difficult stuff that you should approach only when thoroughly familiar with the hardware involved.

An interesting demonstration of the power of **JTerm**'s ring buffer is provided by the help routine **ShowHelp**, which can be invoked at any time from within **JTerm** by pressing the F1 function key. **ShowHelp** works in three stages: It saves the contents of the screen buffer on invocation to the heap; it clears the screen and displays some help information, and finally, once any key is pressed, it reloads the saved screen from the heap and gives control back to **JTerm**'s main loop.

However, while the help screen is displayed, **JTerm**'s main loop isn't paying any attention *at all* to the serial port. If **JTerm** were a polled I/O program, any characters arriving while the help screen was displayed would be lost. But you'll see when the help screen vanishes that any characters that arrived "behind" the help screen are rapidly displayed until the ring buffer is once again empty. Obviously, if you left the help screen on display long enough for 1024 characters to arrive from the remote system, the ring buffer would wrap around and the oldest characters would be overwritten and lost. However, there's nothing that prevents you from boosting the size of the **CircularBuffer** type to 4K, or even 32K characters, if you think you need to.

```
{--------------------------------------------------------------}
{                            JTERM                             }
{                                                              }
{                                  by Jeff Duntemann           }
{                                  Turbo Pascal V5.0           }
{                                  Last update 7/24/88         }
{                                                              }
{ This is an interrupt-driven "dumb terminal" program for the  }
{ PC.  It illustrates the use of Turbo Pascal's INTERRUPT      }
{ procedures, and in a lesser fashion the use of serial port   }
{ hardware.  It is currently hardwired to COM1 for simplicity's}
{ sake.                                                        }
{                                                              }
{                                                              }
{                                                              }
```

```
{-------------------------------------------------------------}


PROGRAM JTerm;

USES DOS,CRT;


CONST
  COM1INT = 12;        { Vector # for COM1: (IRQ4) }

  { 8250 control registers, masks, etc. }
  RBR       = $3F8;    { 8250 Receive Buffer Register      }
  THR       = $3F8;    { 8250 Transmit Holding Register    }
  LCR       = $3FB;    { 8250 Line Control Register        }
  IER       = $3F9;    { 8250 Interrupt Enable Register    }
  MCR       = $3FC;    { 8250 Modem Control Register       }
  LSR       = $3FD;    { 8250 Line Status Register         }
  DLL       = $3F8;    { 8250 Divisor Latch LSB            }
  DLM       = $3F9;    { 8250 Divisor Latch MSB            }
  DLAB      = $80;     { 8250 Divisor Latch Access Bit     }


  BAUD300  = 384;      { Value for 300 baud operation      }
  BAUD1200 = 96;       { Value for 1200 baud operation     }
  NOPARITY = 0;        { Comm format value for no parity   }
  BITS8     = $03;     { Comm format value for 8 bits      }
  DTR       = $01;     { Value for Data Terminal Ready     }
  RTS       = $02;     { value for Ready To Send           }
  OUT2      = $08;     { Bit that enables adapter interrupts }

  { 8259 control registers, masks, etc. }
  OCW1      = $21;     { 8259 Operation Control Word 1     }
  OCW2      = $20;     { 8259 Operation Control Word 2     }
  IRQ4      = $10;     { Mask to turn IRQ4 interrupts on/off }




TYPE
  CircularBuffer = ARRAY[0..1023] OF Char;  { A 1K input buffer }


VAR
  Quit        : Boolean;              { Flag for exiting the program }
  HiBaud      : Boolean;              { True if 1200 baud is being used }
  KeyChar     : Char;                 { Character from keyboard }
  CommChar    : Char;                 { Character from the comm port }
  Divisor     : Word;                 { Divisor value for setting baud rate }
  Clearit     : Byte;                 { Dummy variable }
  Buffer      : CircularBuffer;       { Our incoming character buffer }
  LastRead,                           { Index of the last character read }
  LastSaved   : Integer;              { Index of the last character stored }
  NoShow      : SET OF Char;          { Don't show characters set }
  OldVector   : Pointer;              { Global storage slot for the old }
                                      { interrupt vector }
```

```
{$I SHOWHELP.SRC}  { JTerm's minimal help system }


PROCEDURE EnableInterrupts;

INLINE($FB);



{->>>>Incoming (Interrupt Service Routine)<<<<-----------------}
{                                                              }
{ This is the ISR (interrupt Service Routine) for COM1.  Note: }
{ DO NOT call this routine directly; you'll crash hard.  The   }
{ only way Incoming takes control is when a character coming   }
{ in from the modem triggers a hardware interrupt from the     }
{ serial port chip, the 8250 UART.  Note that the register     }
{ pseudo-parameters are not needed here, and you could omit    }
{ them.  However, omitting them doesn't really get you any     }
{ more speed or reliability.                                   }
{--------------------------------------------------------------}


PROCEDURE Incoming(Flags,CS,IP,AX,BX,CX,DX,SI,DI,DS,ES,BP : Word);
INTERRUPT;

BEGIN
  { Our first job is to enable interrupts during the ISR: }
  EnableInterrupts;
  { The first "real work" we do is either wrap or increment the index }
  { of the last character saved.  If the index is "topped out" at     }
  { 1023, we force it to zero.  This makes the 1024-byte buffer       }
  { "circular," in that once the index hits the end, it rolls over to }
  { the beginning again. }
  IF LastSaved >= 1023 THEN LastSaved := 0 ELSE Inc(LastSaved);

  { Next, we read the actual incoming character from the serial port's}
  { one-byte holding buffer: }
  Buffer[LastSaved] := Char(Port[RBR]);

  { Finally, we must send a control byte to the 8259 interrupt  }
  { controller, telling it that the interrupt is finished:      }
  Port[OCW2] := $20;                     { Send EOI byte to 8259 }
END;



{$F+}
PROCEDURE JTermExitProc;

BEGIN
  Port[IER] := 0;                      { Disable interrupts at 8250 }
  Port[OCW1] := Port[OCW1] OR IRQ4;          { Disable IRQ4 at 8259 }
  Port[MCR] := 0;                        { Bring the comm line down  }
  SetIntVec(Com1Int,OldVector);   { Restore previously saved vector }
END;
{$F-}


```

```

PROCEDURE SetupSerialPort;

BEGIN
  LastRead  := 0;  { Initialize the circular buffer pointers }
  LastSaved := 0;

  Port[IER] := 0;  { Disable interrupts while we're setting them up }

  GetIntVec(Com1Int,OldVector);              { Save old IRQ4 vector }
  ExitProc := @JTermExitProc;           { Hook exit proc into chain }
  SetIntVec(Com1Int,@Incoming); { Put ISR address into vector table }

  Port[LCR] := Port[LCR] OR DLAB;  { Set up 8250 to set baud rate   }
  Port[DLL] := Lo(Divisor);        { Set baud rate divisor          }
  Port[DLM] := Hi(Divisor);
  Port[LCR] := BITS8 OR NOPARITY;      { Set word length and parity }
  Port[MCR] := DTR OR RTS OR OUT2;     { Enable adapter, DTR, & RTS }
  Port[OCW1] := Port[OCW1] AND (NOT IRQ4); { Turn on 8259 IRQ4 ints }
  Clearit := Port[RBR];                { Clear any garbage from RBR }
  Clearit := Port[LSR];                { Clear any garbage from LSR }

  Port[IER] := $01;        { Enable interrupt on received character }
END;


FUNCTION InStat : Boolean;

BEGIN
  IF LastSaved <> LastRead THEN InStat := True
    ELSE InStat := False;
END;


FUNCTION InChar : Char;     { Bring in the next character }
                            {    from the ring buffer }
BEGIN
  IF LastRead >= 1023 THEN LastRead := 0
    ELSE LastRead := Succ(LastRead);
  InChar := Buffer[LastRead];
END;


PROCEDURE OutChar(Ch : Char);   { Send a character to the comm port }

BEGIN
  Port[THR] := Byte(Ch)     { Put character ito Transmit Holding Register }
END;


{>>>>>JTERM MAIN PROGRAM<<<<<}

BEGIN
  HiBaud := True;               { JTerm defaults to 1200 baud; if "300"    }
  Divisor := BAUD1200;          { is entered after "JTERM" on the command }
  IF ParamCount > 0 THEN        { line, then 300 baud is used instead.     }
    IF ParamStr(1) = '300' THEN
      BEGIN
```

```
          HiBaud := False;
          Divisor := BAUD300
        END;

    DirectVideo := True;
    NoShow := [#0,#127];            { Don't display NUL or RUBOUT }
    SetupSerialPort;                { Set up serial port & turn on interrupts }


    ClrScr;
    Writeln('>>>JTERM by Jeff Duntemann');

    Quit := False;         { Exit JTERM when Quit goes to True }
    REPEAT

      IF InStat THEN       { If a character comes in from the modem }
        BEGIN
          CommChar := InChar;                          { Go get character  }
          CommChar := Char(Byte(CommChar) AND $7F);  { Mask off high bit }
          IF NOT (CommChar IN NoShow) THEN            { If we can show it,}
            Write(CommChar)                            {  then show it! }
        END;

      IF KeyPressed THEN  { If a character is typed at the keyboard }
        BEGIN
          KeyChar := ReadKey;         { First, read the keystroke }
          IF KeyChar = Chr(0) THEN  { We have an extended scan code here }
            BEGIN
              KeyChar := ReadKey;   { Read second half of extended code  }
              CASE Ord(KeyChar) OF
               59 : ShowHelp;       { F1 : Display help screen }
              END { CASE }
            END
          ELSE
          CASE Ord(KeyChar) OF
           24 : Quit := True;    { Ctrl-X: Exit JTerm }
           26 : ClrScr;          { Ctrl-Z: Clear the screen }
           ELSE OutChar(KeyChar)
          END;  { CASE }
        END

    UNTIL Quit
  END.
```

JTerm is about as simple an interrupt-driven terminal program as you can write, and gives you all the "rough stuff" for the core of a solid communications application. To add additional commands you need only add new options to the **CASE** statement in the main program loop. The distant descendent of **JTerm** that I use has **XMODEM** file transfer capability, data capture, line by line text file transmission with WordStar high bit/control character removal, an autodialer, and a simple command interpreter for command files, all in Turbo Pascal. As that program runs to about 5600 lines, it will have to wait for another book. But there is nothing there that you cannot do yourself, with a little planning and some low down, machine-level smarts.

23.8: PROCEDURAL TYPES AND PARAMETERS (VERSION 5.0)

In Pascal, we're very used to a strict separation of code and data that rivals Church and State in America. Procedures are what a program does, and variables are little boxes that contain values. Turbo Pascal 5.0 introduces *procedural types,* a new feature that Modula 2 programmers have always had, and with it this separation of code and data blurs a little.

Procedural types may be used to declare procedural variables, which are variables whose values are the names of procedures. Furthermore, procedural variables may be passed to procedures as procedural parameters.

This is a strange notion to the Pascal born and bred, and procedural types explain poorly without using examples all the way. Consider this type definition:

```
TYPE
  HitProc = PROCEDURE(Foundit : SearchRec; Indir : String);
```

Here, **HitProc** is a procedural type. The definition as you see it here actually tells you two things about type **HitProc**:

1. Subprograms declared as type **HitProc** are procedures, rather than functions. Procedural types may also be functions that return values.
2. Subprograms declared as type **HitProc** take two parameters, one of type **SearchRec** (from the **DOS** unit) and the other a default-length string.

In a sense, these two things are the requirements for any subprogram that may be considered of type **HitProc**. Note that the *names* of the formal parameters (i.e, **FoundIt** and **InDir**) are not important. Only their *types* matter. For example, the following simple procedure is of type **HitProc**:

```
PROCEDURE ShowName(FoundFile : SearchRec; ItsDir : String);

BEGIN
  Writeln(ItsDir,'\',FoundFile.Name)
END;
```

It is of type **HitProc** because its procedure header matches the header specified in the **HitProc** procedural type definition: **ShowName** is a procedure, not a function, and its two parameters are the same types as those in the **HitProc** definition, and they are in the same order.

For the same reasons, the following procedure is also of type **HitProc**:

```
PROCEDURE KillFile(Victim : SearchRec; ItsDir : String);

VAR
```

```
  F : File;

BEGIN
  Assign(F,Victim.Name);
  Erase(F)
END;
```

Again, **KillFile**'s procedure header matches the one defined as **HitProc**. The formal parameter names are different, but the types are the same.

Passing Procedures as Parameters

The single greatest use of procedural types is to enable the passing of procedures as parameters to other procedures. Suppose, for a moment, that we had a procedure that searched a directory for files that match a given filespec like *.BAK or JLIST*.PAS. Call this procedure **SearchOne**. Basically, we pass **SearchOne** a directory path and a filespec. What **SearchOne** does is search the directory for files that match the filespec, using the **FindFirst** and **FindNext** procedures from the **DOS** unit, as described in Section 20.9.

Now, the tricky part: Each time **SearchOne** finds a file that matches the filespec, it has to do something. (Otherwise, why bother?) Now, we don't want to hard code some action into **SearchOne** if we can avoid it. We'd like to somehow pass some action to **SearchOne**, for **SearchOne** to take each time it finds a matching file. That way, **SearchOne** becomes a completely general search "engine" that we can apply exactly as we need. At one point in a program, we may want **SearchOne** to do nothing more than print the names of the files it finds. At another point, we might want **SearchOne** to count the number of bytes in each file found, and add the number to a running total. (Ever wonder how many bytes on your hard disk are tied up in a multitide of .BAK files?) Finally, we might want **SearchOne** to delete every .BAK file it finds, to free up some of that space.

Are you beginning to get the drift?

If we create a separate procedure for each action we wish **SearchOne** to take, we can pass the appropriate procedure to **SearchOne** as a procedural parameter. **SearchOne** might be defined this way:

```
PROCEDURE SearchOne(Directory : String;
                    Spec      : String;
                    Attribute : Byte;
                    DoIt      : HitProc);
```

The additional procedure **Attribute** allows us to narrow the search to things like hidden or system files, or to directories. **Attribute** takes a DOS attribute byte, as described in Section 20.7.

The **DoIt** formal parameter is of type **HitProc**. We can pass **SearchOne** the name of any subprogram that matches the **HitProc** spec, in the **DoIt** parameter. If we pass **SearchOne** the **ShowName** procedure given on page 590, **SearchOne** will call **ShowName** each time it makes a *hit* during its search, and the name of the found file will be displayed on the screen. Alternatively, if we pass **SearchOne** the **KillFile** procedure given on page 590, **SearchOne** will delete each file that matches the filespec passed in **Spec**. In either case, all we pass is the *name* of the procedure, but not the names of any parameters:

```
SearchOne('C:\TURBO5','*.BAK',0,ShowName);
```

Because the name of the hit procedure is passed to **SearchOne** at runtime, we can push **SearchOne** off into a unit, along with the **HitProc** definition itself. I've created such a unit, shown below as SEARCHER.PAS. There are three variations on the search engine in unit **Searcher**, each serving the three different types of directory searches:

- **SearchCurrent** searches the current directory of the current volume.
- **SearchOne** searches a specified directory of a specified volume.
- **SearchAll** searches the entire directory tree of the specified volume *beneath* the specified directory. If the specified directory is '\' (the root) the entire volume is searched.

SearchAll is perhaps the most useful of the three. It is very much the guts of the **Locate** program presented in Section 20.9, generalized through procedural parameters and put off into a unit. But while **Locate** was hard-coded to call a procedure named **DisplayData** each time it found a file, **SearchAll** calls whatever procedure you pass in the **DoIt** parameter. **SearchAll**'s use of **FindFirst** and **FindNext** is nearly identical to that of **Locate**, and thus I won't explain it in detail here. To use the routines in unit **Searcher**, you only need to put the name **Searcher** in your **USES** clause.

```
{------------------------------------------------------------}
{                          Searcher                          }
{                                                            }
{              General-purpose file search unit              }
{                                                            }
{                           by Jeff Duntemann                }
{                           Turbo Pascal V5.0                }
{                           Last update 7/25/88              }
{                                                            }
{                                                            }
{                                                            }
{------------------------------------------------------------}

{$F+}
UNIT Searcher;

INTERFACE

USES DOS;

```

```

{ Note that this unit REQUIRES version 5.0 to compile! }

TYPE
  HitProc = PROCEDURE(Foundit : SearchRec; InDirectory : String);

PROCEDURE SearchAll(Directory: String;
                    Spec : String;
                    Attribute : Byte;
                    DoIt : HitProc);

PROCEDURE SearchOne(Directory : String;
                    Spec      : String;
                    Attribute : Byte;
                    Doit      : Hitproc);

PROCEDURE SearchCurrent(Spec : String; Attribute : Byte; Doit : Hitproc);


IMPLEMENTATION


{->>>>SearchAll<<<<------------------------------------------}
{                                                           }
{ UNIT FILE: SEARCHER.PAS  --  Last Modified 6/29/88        }
{                                                           }
{ This is a search "engine" that traverses the entire DOS    }
{ directory tree of the current disk volume, looking for files }
{ that match a filespec passed in Spec, and an attribute byte  }
{ passed in Attribute.  Whenever a matching file is found, the }
{ found file's DTA is passed to a procedural parameter, which  }
{ then takes some action using the information in the DTA.     }
{                                                           }
{ The underlying logic of using FIND FIRST and FIND NEXT is    }
{ almost identical to that of the LOCATE.PAS program, with the }
{ the difference that LOCATE.PAS only displays information on  }
{ the found files.  Passing different procedures in HitProc    }
{ allows SearchAll to perform any action on a found file that  }
{ you care to code up as a procedural parameter.               }
{-----------------------------------------------------------}

PROCEDURE SearchAll(Directory: String;
                    Spec : String;
                    Attribute : Byte;
                    DoIt : HitProc);

VAR
  CurrentDTA : SearchRec;
  TempDirectory,NextDirectory : String;


BEGIN
  { First we look for any subdirectories.  If any are found, }
  { we make a recursive call and search 'em too: }

  { Suppress unnecessary backslashes if we're searching the root: }
  IF Directory = '\' THEN
    TempDirectory := Directory + '*.*'
```

```
      ELSE
        TempDirectory := Directory + '\*.*';

      { Now make the FIND FIRST call for directories: }

      FindFirst(TempDirectory,$10,CurrentDTA);


      { Here's the tricky stuff.  If we get an indication that there is }
      { at least one more subdirectory within the current directory,    }
      { (indicated by lack of error codes 2 or 18) we must search it    }
      { by making a recursive call to SearchDirectory.  We continue     }
      { recursing and returning from the searched subdirectories until  }
      { we get a code indicating none are left. }
      WHILE (DOSError <> 2) AND (DOSError <> 18) DO
        BEGIN
          IF  ((CurrentDTA.Attr AND $10) = $10)   { If it's a directory }
          AND (CurrentDTA.Name[1] <> '.') THEN  { and not '.' or '..' }
            BEGIN
              { Add a slash separating sections of the path if we're not }
              { currently searching the root: }
              IF Directory <> '\' THEN NextDirectory := Directory + '\'
                ELSE NextDirectory := Directory;

              { This begins with the current directory name, and copies }
              { the name of the found directory from the current DTA to }
              { the end of the current directory string.  Then the new  }
              { path is passed to the next recursive instantiation of   }
              { SearchDirectory. }
              NextDirectory := NextDirectory + CurrentDTA.Name;

              { Here's where we call "ourselves." }
              SearchAll(NextDirectory,Spec,Attribute,DoIt);

            END;
           FindNext(CurrentDTA);  { Now we look for more... }
        END;

      { Now we can search for files, once we've run out of directories.  }
      { This is conceptually simpler, as recursion is not involved.       }
      { We combine the path and the file spec into one string, and make  }
      { the FIND FIRST call: }

      { Suppress unnecessary slashes for root search: }
      IF Directory <> '\' THEN
        TempDirectory := Directory + '\' + Spec
      ELSE TempDirectory := Directory + Spec;

      { Now, make the FIND FIRST call: }
      FindFirst(TempDirectory,Attribute,CurrentDTA);

      IF DOSError = 3 THEN       { Bad path error }
        Writeln('Path not found; check spelling.')

      { If we found something in the current directory matching the filespec, }
      { call the procedural parameter to take some action on the found DTA:   }
      ELSE IF (DOSError = 2) OR (DOSError = 18) THEN
        { Null; Directory is empty }
```

```
  ELSE
    BEGIN
      DoIt(CurrentDTA,Directory);         { Call the procedural parameter }
      IF DOSError <> 18 THEN              { More files are out there...   }
        REPEAT
          FindNext(CurrentDTA);           { Look for additional matches   }
          IF DOSError <> 18 THEN          { More entries exist }
          DoIt(CurrentDTA,Directory)      { Call the procedural parameter }
        UNTIL (DOSError = 18) OR (DOSError = 2)  { Ain't no more! }
    END
END;



{->>>>SearchOne<<<<----------------------------------------------}
{                                                                }
{ UNIT FILE: SEARCHER.PAS  --  Last Modified 5/28/88             }
{                                                                }
{ This procedure is a subset of SearchAll, in that it only       }
{ searches the directory specified in Directory, and not the     }
{ entire directory tree of the current disk volume.  In all      }
{ other respects it operates the same way.                       }
{----------------------------------------------------------------}


PROCEDURE SearchOne(Directory : String;
                    Spec      : String;
                    Attribute : Byte;
                    Doit      : Hitproc);

VAR
  TempDirectory : String;
  CurrentDTA : SearchRec;

BEGIN
  { Suppress unnecessary slashes for root search: }
  IF Directory <> '\' THEN
    TempDirectory := Directory + '\' + Spec
  ELSE TempDirectory := Directory + Spec;

  { Now, make the FIND FIRST call: }
  FindFirst(TempDirectory,Attribute,CurrentDTA);

  IF DOSError = 3 THEN         { Bad path error }
    Writeln('Path not found; check spelling.')

  { If we found something in the current directory matching the filespec, }
  { call the procedural parameter to take some action on the found DTA:   }
  ELSE IF (DOSError = 2) OR (DOSError = 18) THEN
    { Null; Directory is empty }
  ELSE
    IF DOSError <> 18 THEN                { More files are out there...   }
      BEGIN
        DoIt(CurrentDTA,Directory);       { Call the procedural parameter }
        REPEAT
          FindNext(CurrentDTA);           { Look for additional matches   }
          IF DOSError <> 18 THEN          { More entries exist }
            DoIt(CurrentDTA,Directory);   { Call the procedural parameter }
        UNTIL (DOSError = 18) OR (DOSError = 2)        { Ain't no more! }
      END
  END;



  {->>>>SearchCurrent<<<<-------------------------------------}
  {                                                           }
  { UNIT FILE: SEARCHER.PAS  --  Last Modified 5/28/88        }
  {                                                           }
  { This procedure uses the same FIND FIRST/FIND NEXT logic of }
  { SearchAll and SearchOne, but only searches the current    }
  { directory.  It therefore does not need to be passed a     }
  { parameter specifying the directory to be searched.        }
  {-----------------------------------------------------------}


  PROCEDURE SearchCurrent(Spec : String; Attribute : Byte; Doit : Hitproc);

  VAR
    Directory : String;

  BEGIN
    GetDir(0,Directory);  { Query DOS for the name of the current directory }
    SearchOne(Directory,Spec,Attribute,DoIt);
  END;


  END.
```

With the **Searcher** unit, you can create any number of useful file utilities with almost no additional code at all. For example, the following short program tallies the number of bytes of space taken by files matching the filespec passed on the command line, throughout the entire disk volume. **Tally** is a global version of **Spacer** (see Section 21.4) that doesn't display file information.

```
{-----------------------------------------------------------}
{                          Tally                            }
{                                                           }
{  File size tally utility; works throughout directory trees }
{                                                           }
{                            by Jeff Duntemann              }
{                            Turbo Pascal V5.0              }
{                            Last update 7/25/88            }
{                                                           }
{                                                           }
{                                                           }
{-----------------------------------------------------------}

{$F+}         { To be safe, use FAR calls throughout... }
PROGRAM Tally;

USES DOS,Searcher;  { Using SEARCHER requires Version 5.0! }

```

```
VAR
  I          : Integer;
  Total      : LongInt;
  SearchSpec : String;
  InitialDirectory : String;


PROCEDURE Tallier(Foundit : SearchRec;InDirectory : String);

BEGIN
  IF InDirectory = '\' THEN
    Writeln(InDirectory,Foundit.Name)
  ELSE
    Writeln(InDirectory,'\',FoundIt.Name);
  Total := Total + Foundit.Size;
END;


BEGIN
  IF ParamCount = 0 THEN
    BEGIN
      Writeln('>>TALLY<<  V1.00  By Jeff Duntemann');
      Writeln('           From the book, COMPLETE TURBO PASCAL 5.0');
      Writeln('           Scott, Foresman & Co. 1988');
      Writeln('           ISBN 0-673-38355-5');
      Writeln;
      Writeln('This program searches for all files matching a given ');
      Writeln('filespec on the current disk device, in any subdirectory.');
      Writeln('It displays their full pathnames, and keeps a total of ');
      Writeln('the size that each occupies, so that you can determine');
      Writeln('just how much space you have tied up in .BAK files,');
      Writeln('throughout your entire disk volume.');
      Writeln;
      Writeln('CALLING SYNTAX:');
      Writeln;
      Writeln('TALLY <filespec>');
      Writeln;
      Writeln('For example, to find out how much space your screen capture');
      Writeln('files (ending in .CAP) occupy, you would enter:');
      Writeln;
      Writeln('TALLY *.CAP');
      Writeln;
    END
  ELSE
    BEGIN
      Total := 0;
      Writeln;
      SearchSpec := ParamStr(1);
      { A "naked" filespec searches the entire volume: }
      IF Pos('\',SearchSpec) = 0 THEN
        InitialDirectory := '\'
      ELSE
        BEGIN
          { This rigamarole separates the filespec from the path: }
          I := Length(SearchSpec);
          WHILE SearchSpec[I] <> '\' DO I := Pred(I);
          InitialDirectory := Copy(SearchSpec,1,I-1);
          Delete(SearchSpec,1,I);
```

```
              END;
           SearchAll(InitialDirectory,SearchSpec,0,Tallier);
           Writeln('The files listed occupy ',Total,' bytes of disk space.');
        END
  END.
```

I won't print any futher code examples (this book has gone on too long already), but I'll offer some suggestions for other uses of the search engines in unit **Searcher**:

- Create a routine that builds a linked list of directory entries using a search engine. Start with the hard-coded list builder routine **GetDirectory** from Section 21.4.
- Write a program that copies specified files from all over a volume (i.e., *.PAS) to another drive unit. This might be one way to backup important files onto a Bernoulli cartridge or a 1.2MB diskette. Be sharp and set the archive attribute bit while you're at it.
- Create a BAK-killer that erases all .BAK files all over a disk volume.
- Build a routine that searches a disk volume for BGI driver files, so that your graphics applications need never crash for not knowing where the drivers are. Add a feature allowing the program to use only the most recent version of a driver if two otherwise identical drivers are found in two different directories on a volume.

With procedural types, you can now write truly general tools that take action based on parameters you pass at runtime. This extraordinarily powerful feature takes some getting used to, but once you understand it you will be creating "engines" of all sorts that expose the "Pascal is a kiddie language; use C" nonsense as the corrosive lie that it has always been.

24 Assembly Language Interface to Turbo Pascal

Given that the whole reason for inventing high level languages like Pascal was to *avoid* having to code in assembler, the current obsession with assembly language interface to Turbo Pascal has to make me smile. It is by far the question I am asked most about Turbo Pascal. The whole issue is a head-scratcher. Assembly language smells too much like work to me; I see it as programming with all the fun squeezed out. But there are people who build ships in bottles; there are people who model entire railroads in Z scale; there are people who sail the Pacific in hollow trees. This section is dedicated to them.

There are actually two very good reasons for writing Turbo Pascal subprograms in assembly code. The first reason is to do something that Turbo Pascal can't do on its own. Many people think this is a long list; after all, Pascal is Pascal. In fact, the list is very short indeed.

Turbo Pascal's suite of extensions for systems programming is nothing short of brutal. You have complete read/write access to every byte in the 8086's megabyte of address space. You have access to every I/O port in the 8086's 64K I/O address range. You can map a variable at any memory address. You can make 8086 software interrupts. You can read the current contents of the CS, DS, or SS registers. You can extract the current segment and offset address of any data item. You have a high speed fill and a high speed block move procedure. You have a host of sneaky tricks for defeating Pascal's strong typing. The point here is this: before you *assume* that something needs to be done in assembly language, think it over once again, keeping the above weaponry in mind.

Here is a list of things that I have identified as requiring some machine code intervention to accomplish at all:

1. Resident utilities.
2. Multitasking.
3. Register peeks and pokes.

That's a short list. With the exception of item 3, it's also a mean one. I'm not going to take on goblins 1 and 3 in this book, since doing so would blow it up to the size of a Sears catalog. A separate book would cover them, and it would *not* be a thin book, (and I'm considering such a book).

Now, the second reason for writing Turbo Pascal subprograms in assembly code is more subjective: To do things that can't be done quickly enough in Pascal itself. Purists and other madmen would say *everything*, but let's be reasonable. My list follows:

1. Joystick interface.
2. Text buffer clearing.
3. Text region (window) moving and scrolling.
4. Fast polled serial communications.
5. Nearly *anything* involving graphics.
6. Matrix math. (Includes spreadsheet recalculation)

Surprise! It's another short list. Actually, item 5 contains a universe of complication that could fill one book or several. The bottom line is that the 8088 processor running at PC speeds just doesn't have the horsepower to do acceptable graphics, even in assembler, so Pascal graphics have to be even worse. They are. Human nature being what it is, *no* machine will ever do graphics quickly enough to suit an unappreciative user, so the best you can do is hand-code the critical routines in assembler and apologize frequently.

If you're working in assembler outside of these areas, you may well be fooling yourself into working too hard. Think again. If you know you're right, share your discovery with me, and I'll add it to my list.

Use an Assembler!

There are two ways to use machine code from within a Turbo Pascal program. One is the **INLINE** statement, which accepts numeric literals as arguments and inserts them into the generated program code at that point, verbatim. There's no protection in it; if the codes tell the CPU to walk off the edge of the world, it will, smiling. **INLINE** does not understand assembler mnemonics like **MOV AX,$F6**. You must hand-assemble your mnemonics into binary numbers like **$B8/$F6** before **INLINE** will accept them. Doing this to more than a handful of mnemonics is ugly, grueling work, prone to many errors and nasty ones. If you intend to do any amount of assembly language interface, and if you value your time at more than 25 cents an hour, a good assembler like Turbo Assembler will pay for itself in two weeks flat.

Actually, if all you want to do is translate short and simple sequences of mnemonics into binary opcodes, DOS DEBUG can do that well enough. But if you're serious enough to work in assembler at all, you should be serious enough to buy the proper tools. All the remaining discussion of non-INLINE assembly-language techniques in this book will assume the use of Turbo Assembler or MASM.

The other method, external machine-code routines, *requires* an assembler, so in that you have no choice.

24.1: WORKING WITH INLINE

The portion of the Turbo Pascal compiler that actually writes out the opcodes that comprise your machine-readable program is called the *code generator.* Code generation is an art approaching black magic, and I don't pretend to know a great deal about it. Ordinarily, the code generator takes its cues from the portion of the compiler that parses and analyses your ASCII source code file, and based on those cues it creates your program code file.

However, there are two instances in which the code generator takes a break and stops generating code. One is at the reserved word **EXTERNAL**, which indicates that code is going to be read from a disk file for awhile. We'll speak of this at length below. The other instance is when it encounters the **INLINE** reserved word. An **INLINE** statement is in some ways like an internally-stored version of an external machine code file. The code generator stops generating code and simply pulls it verbatim from the list of binary numbers you have written as arguments to the **INLINE** statement.

An **INLINE** statement can be dropped in anywhere. It is *not* a procedure call. The statement:

```
INLINE($CD/$05);
```

causes the code generator to pause and insert the two binary numbers $CD and $05 into the code file it's generating for you. Those two numbers are the opcodes for the instruction

```
INT 05
```

which you might recognize as the Print Screen interrupt from the IBM PC ROM BIOS.

The general form of an **INLINE** statement is the following:

```
INLINE(<arg>/<arg>/<arg>/<arg>...<arg>);
```

Each <arg> is an argument to the **INLINE** statement. Most arguments are numbers. A number may be expressed as an integer literal in decimal or hexadecimal form, or it may be a named integer constant. Any other type of literal or constant (real, string, Boolean, whatever) will generate this compiler error:

```
Error 30: Integer constant expected
```

Numeric constants and literals passed as arguments to **INLINE** will be evaluated down to either 1 or 2 binary bytes by the code generator. If they fall into the range 0 through 255, they will generate a single binary byte. Any value greater than or equal to 256 will generate 2 bytes.

For example, consider some integer constants:

```
CONST
  BAR = 291;
  FOO = 11;
```

The constant **FOO** when passed to **INLINE** will generate the single binary byte $0B. **BAR**, on the other hand, will generate the two bytes $23 $01. If that seems suspiciously high for the hex equivalent of 291, remember that integers are stored in memory by the 8086 with the *least* significant byte *first* in memory. What we see for **BAR** is not the hexadecimal number $2301 but the number $0123, which is the hex equivalent of decimal 291. This inversion of common sense makes sense from the CPU's perspective, but humans like you and me will simply have to get used to it.

Identifiers other than integer constants may also be passed as arguments to **INLINE**. Care must be taken here, as the sense of what **INLINE** does with these identifiers is not always the same as what the identifiers do in a normal Pascal program. Variables, for example, do not evaluate out to their values. If you pass a global variable's identifier to **INLINE**, it will be replaced with that variable's offset into the data segment. A typed constant's identifier will also evaluate down to the typed constant's offset into the data segment, which is where typed constants are stored starting with Turbo Pascal V4.0. Remember that whenever you are passing an *offset* to **INLINE**, it will be expressed as *two* bytes, even if the value of the offset is less than 256. If, for example, a global variable **BAS** is passed to **INLINE**, and the offset of **BAS** is $22, **INLINE** will express that offset as $22/$00.

Two additional symbols may be used within **INLINE**: < and >. These are **INLINE**'s "override" operators. Ordinarily, a numeric literal argument to **INLINE** is encoded as 1 byte if it falls in the range 0 through 255, or 2 bytes if it has a value higher than 255. Variable and constant identifiers and program counter references generate 2 bytes.

The override operators allow you to force a single-byte value to be encoded as 2 bytes, or a 2-byte value to be coded as a single byte. The < operator instructs the code generator to encode only the least significant byte of a 2-byte argument. The > operator instructs the code generator to encode a single byte argument as 2 bytes, by adding a 0 byte after the single-byte value.

Examples:

```
INLINE($1122/<$3344);   { Generates $22 $11 $44 }
INLINE($5566/>$77);     { Generates $66 $55 $77 $00 }
```

Keep in mind the inversion of the byte order of 2-byte arguments to **INLINE**; this is the reason $1122 is represented in memory as $22 $11. Again, the override operators of **INLINE** are not likely to be useful in a great many situations, but do be aware of them. An example of the override operator < in use can be seen in the **Is87There** function given below:

```
{->>>>Is87There<<<<-------------------------------------------}
{                                                              }
{ Filename: 87THERE.SRC -- Last modified 7/13/88               }
{                                                              }
{ This routine detects the presence of a math coprocessor by   }
{ attempting to initialize the coprocessor.  If something      }
{ meaningful comes back, the coprocessor is present.           }
{                                                              }
{                                                              }
{                                                              }
{--------------------------------------------------------------}

FUNCTION Is87There : Boolean;

VAR
```

```
    ControlWord : Word;

  BEGIN
    ControlWord := 0;  { Clear control word storage to 0 }
    INLINE
      ($90/$DB/$E3/                    { FNINIT }
       $90/$D9/$7E/<ControlWord);      { FNSTCW ControlWord }
    IF Hi(ControlWord) = 0 THEN Is87There := False
      ELSE Is87There := True
  END;
```

Register Peeks and Pokes

Turbo Pascal allows relatively little direct access to the 8086 registers. There are functions to return the current values of segment registers DS, CS, and SS (**DSeg**, **CSeg**, and **SSeg**) and that's all. Going right to the registers is not something you need to do frequently, but there are some arcane occasions that do call for it. The **MSDOS** and **Intr** routines from the **DOS** unit (see Chapter 20) return just about all the 8086 registers, but you may wish to inspect or change the registers without doing either a DOS call or a software interrupt. Doing a dump of the stack from within a function or procedure is one good example that we'll go through in detail a little later.

INLINE is tailor-made for this sort of thing. Four or five binary bytes in an **INLINE** statement will allow you to peek or poke just about any programmer-accessible 8086 register.

The *modus operandi* is simple. Define a global variable which is to contain the returned register value. There's nothing predefined or sacred about the names of the 8086 registers in Turbo Pascal. You can define identifiers **AX**, **BX**, **CX**, **DX**, and all the others, just as given in the 8086 documentation. As most 8086 registers are 16 bits wide, type **Word** will hold them quite well. If you need to access register halves in addition to the register as a whole, you can use the **Hi** and **Lo** built-in functions to return the high 8 bits of the register or the low 8 bits, respectively.

Make sure that the variables you use for working with 8086 registers are *global* variables, defined in the main program variable definition part, and not within one of the procedures or functions. The methods described in this section work *only* with global variables. It is certainly possible to return register values to local variables within a Pascal subprogram, but the methods will not be so simple as with global variables and will require some considerable understanding of the various addressing modes the 8088 offers, particularly those involving indirect references offset from register BP.

Assume for simplicity's sake throughout this entire section that you have defined a suite of global **Word** variables like so:

```
VAR
  AX,BX,CX,DX,BP,SI,DI,ES,SP,FLAGS : Word;
```

The simplest case is the return of 8086 register AX. The following **INLINE** statement will do it:

```
INLINE($A3/AX);         {MOV DS:<var>,AX}
```

One of the arguments to **INLINE** is a binary number. The other is the name of variable **AX**. When Turbo Pascal compiles this statement, it replaces the identifier **AX** with variable **AX**'s offset relative to the data segment register, DS. $A3 is a special form of the 8086 MOV instruction that assumes that DS is acting as a segment register and that an immediate offset is given as part of the instruction. In other words, once Turbo Pascal compiles this statement, the statement might have been expanded to:

```
$A3/$14/$00
```

The $14/$00 is actually an immediate offset supplied by the compiler, representing variable **AX**'s position in the data segment. The value itself may differ from $14; I use this one as an example. The $00 was added since $A3 requires a 16-bit offset, so rather than simply giving an offset of $14, Turbo Pascal dutifully supplies an offset of $0014. The $00 comes *after* the $14 because the 8086 stores 16-bit quantities low-byte first, followed by the high-byte. Yes, it's screwy; but as Intel will tell you again and again, the 8086 architecture was *not* designed to make it easy on assembly-language programmers.

The $A3 opcode is a special case in that it operates only on register AX. We can't use it directly to return any of the other registers. However, moving one register into another takes only four clock cycles, so for the rest of the 8086's registers, all we need to do is move the other register values into AX, and then move the contents of AX into our other variables defined for the other registers. The additional time required is minute, and it makes coding up the rest of our register peeks fairly easy.

The general purpose register-register move opcode is $89, but it requires a second opcode byte to tell the CPU which register is being moved to which register. The complete, 2-byte opcode to move register BX to register AX is $89/$D8. Adding this ahead of our previous code for returning the value of register AX, gives us a complete register peek statement for register BX:

```
INLINE($89/$D8/$A3/BX);  {MOV AX,BX; MOV DS:<var>,AX}
```

$89/$D8 moves the contents of register BX into register AX, and then $A3/BX moves the contents of register AX into the global variable **BX**.

Pushing and Popping the Stack

This, of course, destroys whatever might previously have been in register AX. If protecting the current value of AX is important, you can temporarily push AX's current value onto the system stack, perform the peek, and then pop AX's contents back off the stack into the AX register. This adds considerably to the number of machine cycles used, but we're still taking a very small handful of milliseconds at best. Adding the opcodes for push and pop expands our **INLINE** statement like this:

```
INLINE($50/$89/$D8/$A3/BX/$58)
```

Here, $50 is the PUSH AX opcode, and $58 is the POP AX opcode.

Push and pop operations are also helpful when you want to peek the 8086 flags. The flags reside in a 16-bit word, of which only 9 bits are significant. Each bit is a flag, and each flag is set or cleared under some specific machine conditions, often as the result of executing some 8086 opcode. Again, a good book on 8086 assembly language will tell you what each individual flag stands for.

There is no instruction for moving the flag word into a register. There is, however, an instruction that pushes the flag word onto the stack: PUSHF. We've already examined the POP AX instruction, which pops a word off the top of the stack into AX. Executing PUSHF followed by POP AX pushes the flags onto the stack and then pops them off into AX, effectively moving the flag word into AX. Because pushing or popping the stack involves access to system memory, it takes a lot more time than moving a value from one internal 8086 register into another, but we're still speaking of scant milliseconds here.

This can still be done without losing the current contents of AX by pushing AX onto the stack first, then pushing the flags, popping the flags into AX, moving AX into a global variable, and then finally popping the original contents of AX back off the stack into AX. It can be done like so:

```
INLINE($50/                    { PUSH AX }
       $9C/                    { PUSHF }
       $58/                    { POP AX }
       $A3/FLAGS/              { MOV <var>,AX }
       $58);                   { POP AX }
```

Notice that this **INLINE** statement has been stacked vertically rather than run out on a single line, as the earlier examples were. This was done to allow comments to the left of each group of binary bytes that represents a machine instruction. The comments can, of course, be anything at all, but good practice for **INLINE** is to use the assembly language mnemonic within a comment beside each binary machine instruction. For **INLINE** statements of more than 4 or 5 bytes, this is essential. You may remember what all those binary numbers do *now*, but come back in 6 weeks and have another look. . . . Make it easy on yourself. Comment heavily, and consistently.

Accessing Typed Constants

Typed constants, like global variables, are stored in the data segment, and are accessed the same way as global variables. In fact, it is best to consider them global variables that are given an initial value on program startup.

Turbo Pascal 3.0 and earlier stored typed constants in the code segment rather than the data segment, so there was some extra complication in accessing typed constants from within an **INLINE** statement. No more. Treat byte-, word-, or pointer-sized typed constants exactly as you would a global variable of the same size.

Jumps Within INLINE

Turbo Pascal's **INLINE** feature does little or nothing to make jumps easy to write. You get nothing like the labels available in a real assembler. All jumps have to be followed by a displacement that you must calculate manually, and if you miss it by even a single byte, you could be reaching for the power switch.

Jumps should therefore be handled with some care. Particularly, dropping new code into an **INLINE** statement that contains jumps can throw all your displacements off, forcing you to recalculate them every time you change the number of bytes between a jump instruction and its destination. Having been through this a few times, my recommendation is that once your **INLINE** statement gets complicated enough to start handing you that kind of grief, convert it to an external routine and begin using an assembler.

There are several different flavors of jump instruction in the 8088 instruction set. The two main groups are *long* and *short* jumps. Long jumps require both a segment and an offset address, and will therefore not only take you out of the **INLINE** statement but outside of the current code segment. As each linkable program module in a Turbo Pascal program has its own segment, and because the code within a module cannot be in more than one segment, using long jumps within an **INLINE** statement is unnecessary.

Short jumps will do perfectly well. A short jump remains within the current code segment. Short jumps come in two flavors of their own: Conditional and unconditional. An unconditional jump is like Pascal's **GOTO**; execution moves to the destination and that's that. Conditional jumps perform the jump only if some condition is met.

All short jumps specify their destinations by way of a displacement value. *This is not an offset into the current code segment!* A displacement is the number of bytes away from the current position of the program counter, to which execution will pass. It can be either negative (for backward jumps) or positive.

To further complicate matters, all conditional jumps are limited to 128 bytes forward or 128 bytes backward, since they can only accept a single byte displacement. Only the unconditional short jump can use a 16 bit displacement and move 32K bytes ahead or back.

Expressing negative numbers (which includes negative displacements) in binary uses an interesting notation called *two's complement.* A binary number added to its two's complement yields zero. The best way to put it across is in a table. Study Table 24.1 below for a moment until the scheme seems natural to you.

This table should obviate any need you may have to calculate two's complements on your own; if your **INLINE** statements ever require a backwards jump longer than 64 bytes, you are a brave soul indeed.

Here is an example of a backwards jump coded in **INLINE**:

Table 24.1
Two's Complements

− 1 = $FF	−17 = $EF	−33 = $DF	−49 = $CF
− 2 = $FE	−18 = $EE	−34 = $DE	−50 = $CE
− 3 = $FD	−19 = $ED	−35 = $DD	−51 = $CD
− 4 = $FC	−20 = $EC	−36 = $DC	−52 = $CC
− 5 = $FB	−21 = $EB	−37 = $DB	−53 = $CB
− 6 = $FA	−22 = $EA	−38 = $DA	−54 = $CA
− 7 = $F9	−23 = $E9	−39 = $D9	−55 = $C9
− 8 = $F8	−24 = $E8	−40 = $D8	−56 = $C8
− 9 = $F7	−25 = $E7	−41 = $D7	−57 = $C7
−10 = $F6	−26 = $E6	−42 = $D6	−58 = $C6
−11 = $F5	−27 = $E5	−43 = $D5	−59 = $C5
−12 = $F4	−28 = $E4	−44 = $D4	−60 = $C4
−13 = $F3	−29 = $E3	−45 = $D3	−61 = $C3
−14 = $F2	−30 = $E2	−46 = $D2	−62 = $C2
−15 = $F1	−31 = $E1	−47 = $D1	−63 = $C1
−16 = $F0	−32 = $E0	−48 = $D0	−64 = $C0

```
INLINE($31/$C0/          {XOR AX,AX}
       $B9/$64/$00/      {MOV CX,100}
       $81/$C0/$03/$00/  {ADD AX,3}
       $49/              {DEC CX}
       $75/$F9);         {JNZ -7}
```

The code itself doesn't do anything useful; I chose it because it was easy to understand and follow. The first line XORs the AX register against itself. This is probably the first performance "tweak" assembly language programmers ever learn; what it does is clear AX to zero in three machine cycles; the equivalent MOV AX,0 takes four machine cycles. Well, in some circles every cycle counts.

The next line loads 100 into CX as immediate data. This will become the counter for a loop we're going to assemble. The third opcode does some "work;" it adds three to the AX register. The next line decrements CX, which in this case is acting as a "count" register. It was loaded originally with the number of times we wish to execute this particular loop, and with each pass through the loop it gets decremented by one.

The last line is the jump instruction. It is a *Jump Not Zero,* which means that it checks the state of an 8086 flag called the *zero flag* and makes its decision as to jump or not to jump based on the state of that flag. The decrement instruction that comes just before the jump affects the zero flag; if in decrementing CX, CX goes to zero, the zero flag becomes true.

JNZ will jump as long as the zero flag is not true. The zero flag is set to false each time CX is decremented but does not go to zero. So as long as there remains a nonzero value in CX, JNZ will make its backwards jump by 7 bytes. But as soon as CX goes to zero, the zero flag becomes true, and JNZ will not jump, but simply let control "fall through" to the next instruction. In this case, there are no more instructions in the **INLINE** statement, so the next instruction in the Pascal program takes over.

The only tricky part of setting this up is deciding exactly what displacement value to use for JNZ. You have to count backwards from the location immediately after the *displacement value.* Count 7 bytes backwards from just past the $F9 displacement byte, and you'll find yourself at the beginning of the ADD instruction, which is where you want to be.

The easiest mistake to make in coding jumps like this is to count from after the jump opcode, rather than after the displacement value. This is especially true if you haven't yet figured out the displacement and haven't written down a value for it in its proper place. Don't ever forget: The displacement byte is part of the jump instruction! By the time the displacement is added to the instruction pointer, the instruction pointer is already pointing to the first byte past the end of the instruction. So even if you don't yet know what value the displacement is going to be, fill the byte position with a $00 so that you can be sure you're counting it as you figure the displacement.

The second easiest mistake is to insert an instruction in the middle of a sequence of **INLINE** instructions without adjusting any affected jump displacements. Jumping into the middle of a multibyte instruction is fatal more often than not. Every time you add instructions to or remove instructions from an **INLINE** statement with jumps, check those displacements!

There's very little else to do in coding jumps, but that nasty business of calculating correct displacements makes it thoroughly unpleasant work. Minimize jumps in **INLINE** code, or start using that assembler!

24.2: FUNCTIONS, PROCEDURES, AND LOCAL VARIABLES

Most of the trick in making use of variables from machine code is knowing where they are. As we've seen, global variables and typed constants are easy to pin down: they exist at one unchanging offset from the DS register. When we begin talking about subprograms (procedures and functions) the whole game changes shape drastically.

It has to do with the nature of subprograms in Pascal. Although a subprogram's code is created at compile time and always exists somewhere in the code segment, a subprogram's parameters and data literally *do not exist* until the subprogram is invoked. No space is reserved in the data segment for a subprogram's parameters, nor for its local variables. These critters come into being when a subprogram is called, and they are created on the stack.

It is helpful to remember at this point that the 8088 CPU decrements the stack pointer *first* and then stores the pushed value at the stack location. This means that the stack pointer always points to real data, *not* to unused empty space. Not all CPU's work this way; some store information at the stack location first and then decrement, leaving the stack pointer pointing at free space. Think: first decrement, *then* store, and you won't get confused.

When a subprogram is invoked, Turbo Pascal's runtime code *instantiates* the subprogram on the stack. It can best be explained by using an actual example. Consider these declarations:

```
TYPE
  String30 = String[30];
  IntArray = ARRAY[0..99] OF Integer;


FUNCTION Dummy(VAR Values   : IntArray;
                   Fudge    : Integer;
                   Level,Clearance   : Char;
                   Message : String30) : Boolean;

VAR
  Grade   : Char;
  Cutoff : Char;

BEGIN
  Grade := 'A';
  Cutoff := 'Q';
END;
```

This is not intended to be a "useful" function declaration. It was defined with the parameters and data items shown to make it interesting on the dissection table. Let's follow what happens when **Dummy** is invoked, step by step. As virtually all these steps involve the stack, a diagram of the condition of the stack *after* instantiation is given as Figure 24.1. If you get confused while reading, follow along on the diagram as well.

1. The parameters are pushed on the stack. In our example, **Values** is pushed first. **Values** is a **VAR** parameter, meaning that what the subprogram gets is not a copy of **Values**, but the *address* of **Values**. What is pushed onto the stack is, essentially, a pointer to the actual parameter passed to **Dummy** as **Values**. The 16-bit segment part of the address is pushed first, followed by the 16-bit offset part of the address.

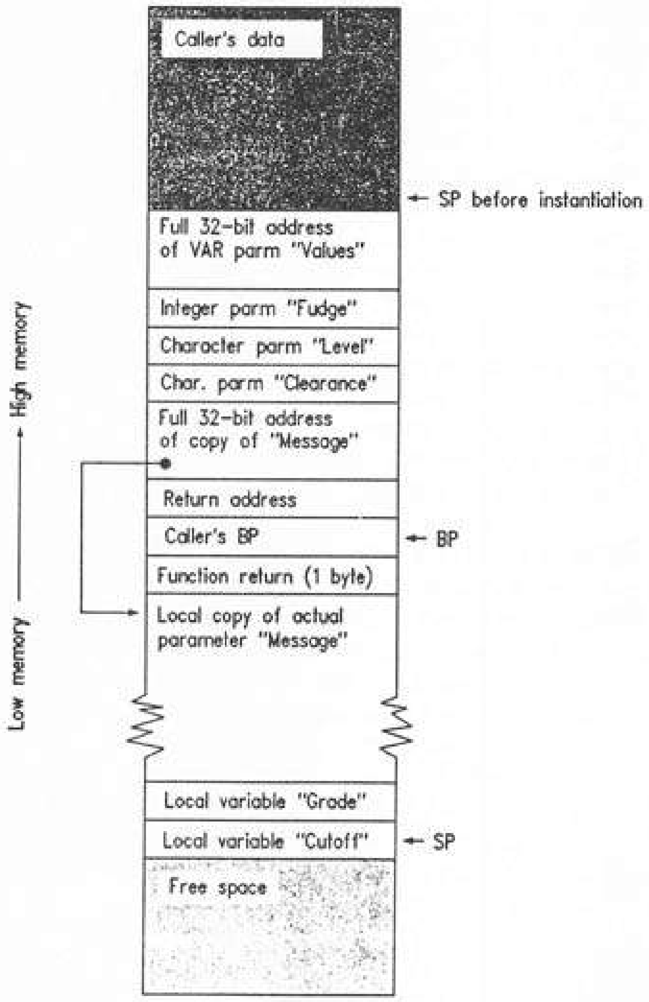
   Remember that we're starting from high memory and moving toward low memory with each item pushed on the stack.

   Parameter **Fudge** is pushed on the stack next. **Fudge** is a value parameter, so a copy of the actual parameter is pushed on the stack. This copy is, like all integers, represented as two bytes. The most significant byte of the integer value is higher in memory. In other words, if at invocation time **Fudge** is passed a value of 17,642 (hex $44EA) the hex byte $44 will be located one byte higher on the stack than the byte $EA.

   The next two parameters, **Level** and **Clearance**, are both characters. Now, while characters are represented as a single byte in Pascal, they are pushed onto the stack as words. Of the 2 bytes pushed onto the stack for each character, the byte higher in memory is a 0 byte ($00) followed by the byte containing the actual character value.

Figure 24.1

Instantiation of a Subprogram



Finally, string parameter **Message** is pushed onto the stack. Message is a value parameter, so you would expect that, like **Fudge**, **Level**, and **Clearance**, a complete copy of the string would be pushed onto the stack. Turbo Pascal 3.0 worked that way, *but version 4.0 and higher do not.* What happens is that a full 4-byte address is pushed onto the stack in lieu of **Message** itself, as though **Message** were a **VAR** parameter. This address is the address of a *copy* of **Message** that is pushed onto the stack later on, as we'll see in step 4.

2. All the parameters are now on the stack. The return address is now pushed onto the stack. This address can be either a full, 32-bit address or simply a 16-bit offset into the code segment. Which one you'll find on the stack is determined by whether the subprogram was compiled as *near* or *far*. Near subprograms use a RET (Return) opcode that can only return control to a 16-bit offset within its own code segment. Far subprograms use a different RET opcode that can return control to a full 32-bit address anywhere in 8086 memory. This means that near subprograms can only be called from within their own code segment, but far subprograms can be called from anywhere at all.

   The default and most frequent case is a near subprogram. Near subprograms are callable only from within their own module. Subprograms in the main program block are near because only the main program and its subprograms can call them. Subprograms in other units cannot call subprograms in the main program block. The same holds true for a unit's "private" subprograms, declared completely within the implementation section of their unit. These private subprograms are not visible outside of their own unit, and therefore are created as near subprograms.

   The subprograms declared in the interface portion of a unit are far subprograms. Also, you may specify that any subprogram anywhere be compiled as a far subprogram by using the {**$F+**} compiler directive.

   As with all addresses pushed onto the stack by Turbo Pascal, the high-order word of the return address is pushed first, followed by the low-order word of the address. Within each of the 2 address words pushed onto the stack, the high-order byte is pushed first, followed by the low order byte.

   There is a conceptual break at this point. Everything so far is done by the Turbo Pascal runtime code, whether the subprogram being instantiated is written in Pascal or declared as an external machine code file. From here on, however, external machine code subprograms are on their own.

3. The stack pointer SP now points to the least significant byte of the return address. To allow easy access to the parameters now on the stack, the stack pointer must be copied into the base pointer BP. This will free the stack pointer to run up and down as the code requires during subprogram execution while still providing a fixed base from which to access parameters.

   The Turbo Pascal runtime code, however, requires that subprograms leave certain registers alone when they execute. These registers are BP, CS, DS, and SS. The reasons for leaving the segment registers CS, DS, and SS alone should be obvious: These registers are literally the only way the program code keeps track of where its code, data, and stack are. The runtime has its own uses for BP, even when the main program is executing and variables are allocated in the data segment rather than on the stack. Also, subprograms can have their own local subprograms, and if a subprogram is executed *within* another subprogram, the outer subprogram depends on BP to keep track of its local variables and parameters. If the inner subprogram loaded its own value into BP without saving the outer program's copy of BP, the outer subprogram would no longer know where its parameters and local variables were once it resumed control.

So before loading the stack pointer into BP, the previous value of BP must itself be saved on the stack. Once BP has been pushed onto the stack, SP can be loaded into BP, providing an anchor point for access to items stored previously on the stack, and allowing the stack to go on and serve the executing code as needed.

In subprograms written in Pascal, the code generator does all this. In your own external machine-code subprograms, you will have to explicitly push BP and move the stack pointer into BP. It's simple enough:

```
PUSH   BP
MOV    BP,SP
```

At this point, both BP and SP point to the old value of BP, now safely ensconced on the stack.

4. The function result is allocated on the stack. If a subprogram is a function, Turbo Pascal needs a location to put a temporary value that will become the function result which the function will return to the calling logic. This is pushed onto the stack immediately after the caller's BP. For all types of functions but string functions, space is allocated on the stack for the full value. In other words, for real number functions, 6 bytes are allocated on the stack for the function result. For pointer functions, 4 bytes are allocated. For integer or word functions, 2 bytes are allocated, and for character, integer, or enumerated type functions, a single byte is allocated. For string functions, the full 32-bit address of a work area elsewhere in memory is pushed onto the stack; the string itself is not pushed.
5. Copies of "oversized" value parameters are allocated on the stack. Strings, records, and arrays passed by value to a subprogram are "private" copies belonging to that subprogram. The subprogram can fold, spindle, or mutilate value parameters at will without affecting the actual parameters elsewhere in the program. This being the case, a physical copy of each actual parameter has to be created somewhere, and that somewhere is on the stack.

   As we mentioned earlier in step 1, copies of things like strings, sets, records, and arrays are not automatically created on the stack when the other, smaller parameters are pushed on the stack. Instead, Turbo Pascal passes a 32-bit address of a copy on the stack. This address, as it turns out, points to a true copy of the actual parameter a little bit further down the stack.

   Our example is no exception. A full copy of **Message** is created on the stack just below the function result. **Message** is a **String30**, meaning that it consists of one length byte plus thirty character data bytes. String values are pushed onto the stack "backwards": The *last* character in the string is pushed first, followed by the second-to-last, and so on, until finally the first character is pushed onto the stack. The very last byte of the string pushed onto the stack is the length byte, element 0 of the string variable. Keep in mind that although individual character variables are pushed onto the stack as words (as described earlier), the characters comprising a string are pushed onto the stack as bytes. This is a slower process, but it saves valuable stack space.

6. Finally, local variables must be allocated on the stack. In subprograms written in Pascal, of course, this happens invisibly. Interesting if invisible things happen here. In tracing code generated by Turbo Pascal, I have found that the runtime code allocates single bytes on the stack for variables like characters, bytes, and Booleans that only require single bytes. Ordinarily, space on the stack is only allocated in chunks of 2 bytes (1 word) at a time. The **PUSH** and **POP** 8088 instructions only manipulate words, not bytes. The Turbo Pascal runtime, however, fakes single byte pushes by performing a **DEC SP** instruction which moves the stack pointer down by one byte. As there is no initial value to push into a local variable, this is all that is required—a **DEC SP** instruction allocates 1 byte on the stack, and that is all that is required.

   I only said a little about it in previous paragraphs to avoid confusion, but this same method is used when allocating string parameters on the stack. The length byte and the individual data characters in a string are all single bytes, and they are pushed individually onto the stack by a process of decrementing SP by one and then storing the bytes on the stack with a **MOV** instruction rather than a **PUSH** instruction.

   Function results are also allocated on the stack this way, so that types represented as single bytes are allocated on the stack as single bytes when they are function results, even though they are allocated as two bytes if they are value parameters.

   If your external machine-code subprograms require local workspace, you must allocate it yourself, by pushing an appropriate number of words onto the stack, or by subtracting the number of bytes needed from the value in the stack pointer:

```
SUB SP,WORKSPACE_SIZE
```

   If **WORKSPACE_SIZE** had earlier been equated to 12, a total of 12 bytes of memory would have been allocated on the stack.

   At this point, all initial work connected with the stack is finished, and the subprogram gets on with its real work. When that work is completed, however, what was done to the stack must be undone for control to return to the calling logic in an orderly fashion.

   Cleaning up the stack is not difficult. The first step is to move the subprogram's current value of BP into the stack pointer SP:

```
MOV SP,BP
```

   If you recall, BP had been left pointing to the calling logic's older copy of BP on the stack. The above instruction brings SP up to that point, in effect destroying any local variables and temporary workspace that had been allocated on the stack. Next, the calling logic's copy of BP is restored by popping it off the stack:

```
POP BP
```

This instruction takes whatever lies at SS:SP and moves it into the BP register. Since SP points to the old copy of BP, that copy of BP is restored into the BP register. Now BP is no longer in the service of the subprogram, but has gone back to the control of the calling logic. The stack pointer, however, now points to the least significant byte of the return address. All remaining cleanup is combined with the act of returning control to the calling logic, with one single instruction:

```
RET <bytes>
```

The **<bytes>** parameter is the exact number of bytes of the subprogram's data remaining on the stack, *not* including the return address. For our example function **Dummy** this would be 14 bytes for all the parameters, keeping in mind that **Message** was passed not as a full copy but as a pointer to a copy that was allocated further down the stack.

Peeking at the Stack

You don't have to be content with inferences in working out the contents of the stack during a function or procedure call. You can go right in and *look* at it if you like.

Recall from the previous section that during subprogram instantiation the Turbo Pascal runtime code pushes parameters on the stack, and then copies the stack pointer register SP into the base pointer register BP. This allows the runtime to use the stack pointer for other things during subprogram execution while "keeping its finger" at the point in the stack where parameters are stored. BP does not change at any point later in the subprogram (unless you explicitly change it, which is not a good idea), so it can be used as a "base" from which to reference parameters on the stack.

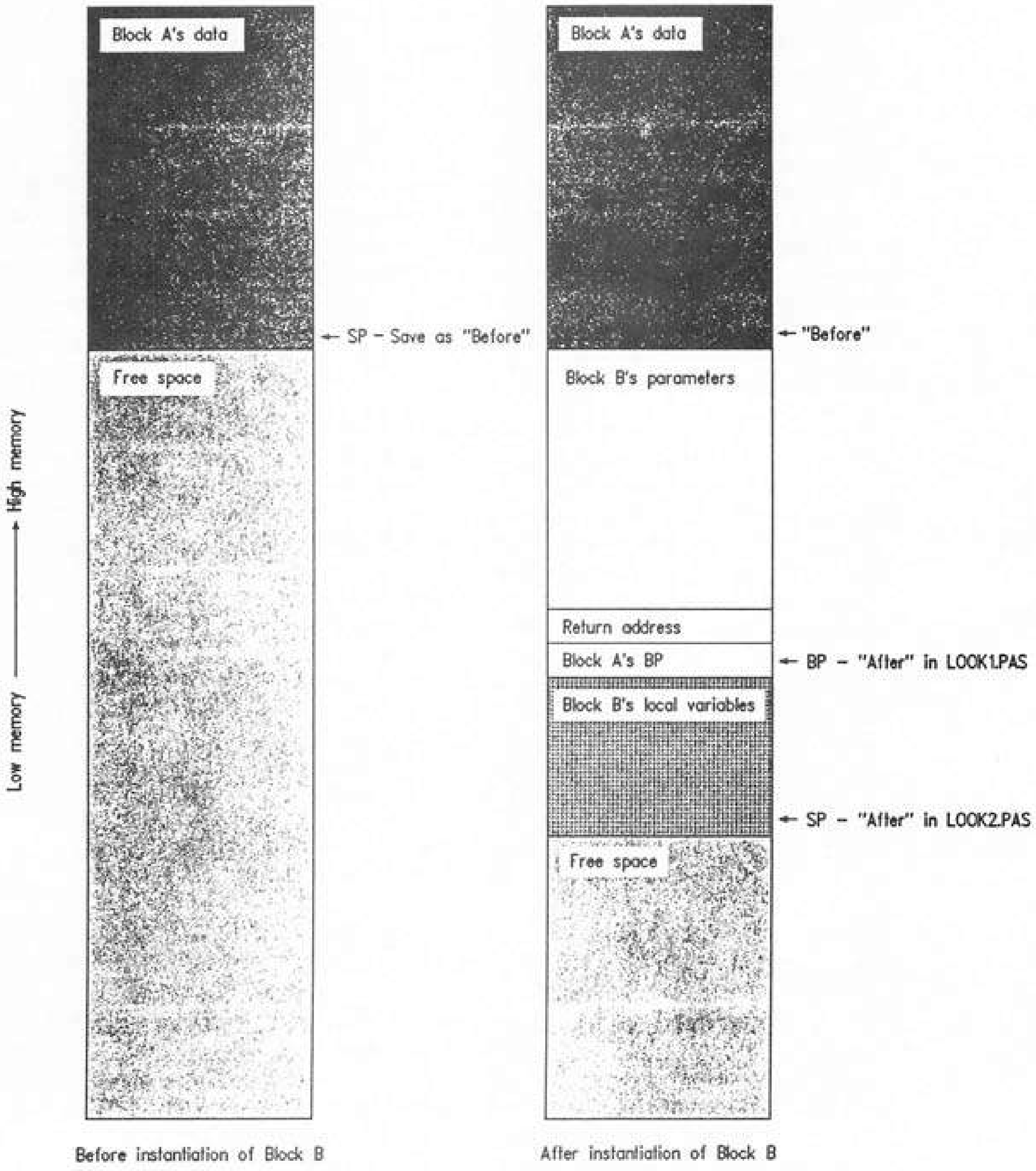
The value of the stack segment register SS is always accessible through the Turbo Pascal built-in function **SSeg**. So looking at parameters on the stack is as simple as obtaining the value in BP, combining the return value from **SSeg** with the value of BP into a pointer, and passing that pointer to the **VarDump** procedure described in Section 20.3. Obtaining the value of BP is done with a short **INLINE** statement:

```
INLINE($8B/$C5/        { MOV AX,BP }
       $A3/Register}; { MOV Register,AX }
```

Creating the pointer can be done with Turbo Pascal's **Ptr** function, which returns a pointer given two word values, one specifying the segment part of the pointer and the other the offset part of the pointer.

VarDump also requires a size parameter to tell it how many bytes to dump. For simplicity's sake you can simply plug in a value equal to the combined size of all your parameters. In other words, if you add up the sizes of all parameters you're passing, simply hard-code that figure into **VarDump**'s **ItSize** parameter.

Figure 24.2
Peeking at the Stack


Block A's data
Free space
SP – Save as "Before"
Low memory ——→ High memory
Block A's data
"Before"
Block B's parameters
Return address
Block A's BP
BP – "After" in LOOK1.PAS
Block B's local variables
SP – "After" in LOOK2.PAS
Free space
Before instantiation of Block B
After instantiation of Block B

You can also allow the machine to calculate how many bytes of parameter are actually on the stack, and dump only that many bytes. It involves a little more coding, but I think it's a lot more satisfying to let computers do what they do best. Consider that *just before* a subprogram is called, the SP register is pointing to the *last* byte allocated on the stack by the Turbo Pascal runtime code for whatever block that had control when the subprogram was called. When the subprogram is called, parameters are allocated on the stack from that point downward. When the subprogram begins executing, the BP register points to the last byte of parameter data allocated for the subprogram.

Figure 24.2 shows a before and after view of the stack when a subprogram is called. The calling logic is a block we'll call Block A, and the subprogram is a block called Block B. Before Block B is called, the stack pointer points to the last byte of data local to Block A. Beneath SP is free space, available for stack growth. SP can be captured at this point as a *before* pointer for the stack peeking trick. You have two choices for the *after* value, as described below.

By "saving" the value of the stack pointer just before the subprogram is called (a *before* value) and then saving BP after the subprogram begins running (an *after* value) the number of bytes of parameters pushed onto the stack for the subprogram will be the difference between the *before* and *after* values. This value can then be given to **VarDump**'s **ItSize** parameter, and **VarDump** will always dump exactly the correct number of bytes of parameter data from the stack, no more, no less.

The short program **Look1** dumps the parameter data from function **Dummy** that we described previously. Figure 24.3 annotates the first of the two dumps of the stack which **Look1** creates and relates it to the parameter list for **Dummy**. The dump shows the state of the parameter stack immediately upon execution of **Dummy**.

```
PROGRAM Look1;

TYPE
  String30 = String[30];
  IntArray = ARRAY[0..99] OF Integer;

VAR
  OK           : Boolean;
  Register     : Word;
  StackMarker  : Pointer;
  Before,After : Word;
  MyArray      : IntArray;

{$I WRITEHEX.SRC}
{$I VARDUMP.SRC}



FUNCTION Dummy(VAR Values   : IntArray;
                   Fudge    : Integer;
                   Level,Clearance : Char;
                   Message  : String30) : Boolean;

VAR
  Grade,Cutoff : Char;
```

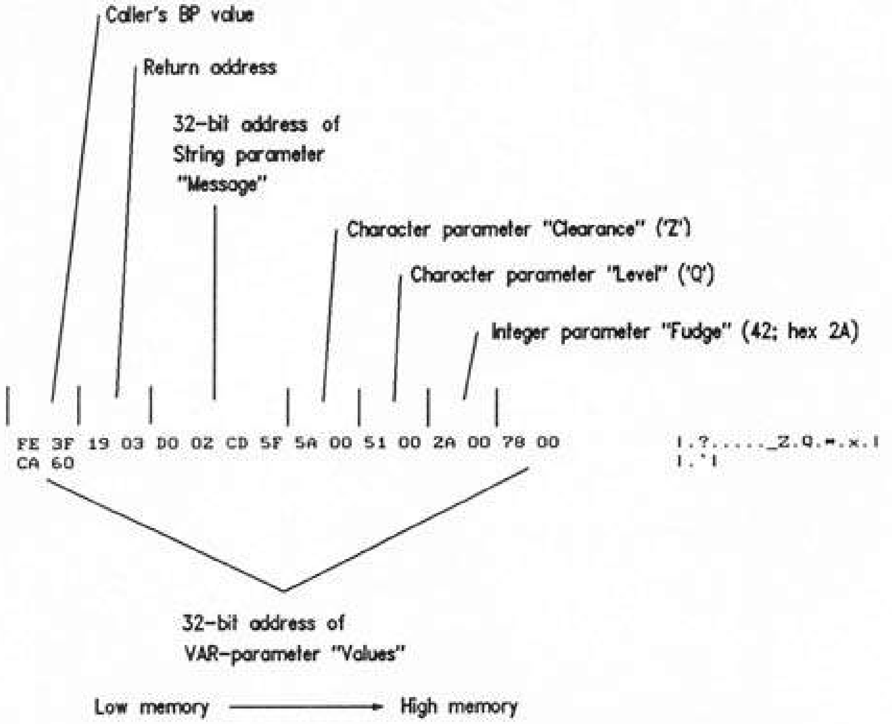
```

BEGIN
  INLINE($90/$90/$90/$90);
  INLINE($8B/$C5/$A3/Register);          { Save BP into Register }
  After := Register;
  StackMarker := Ptr(SSeg,Register);     { Make a pointer SS : BP   }
  VarDump(Output,StackMarker^,Trunc(Before-After));    { Dump stack }
END;



BEGIN
  INLINE($8B/$C4/$A3/Register);        { Save BP into Register }
  Before := Register;
  OK := Dummy(MyArray,42,'Q','Z','I was born on a pirate ship.  ');
END.
```

Figure 24.3

LOOK1's Stack

```
FUNCTION Dummy(VAR Values    : IntArray;
                   Fudge     : Integer;
                   Level,Clearance : Char;
                   Message   : String30) : Boolean;
```

When function "Dummy" is instantiated, its stack looks like this:


Caller's BP value
Return address
32-bit address of String parameter "Message"
Character parameter "Clearance" ('Z')
Character parameter "Level" ('Q')
Integer parameter "Fudge" (42; hex 2A)
32-bit address of VAR-parameter "Values"
Low memory → High memory

```
FE 3F 19 03 D0 02 CD 5F 5A 00 51 00 2A 00 78 00        |.?....._Z.Q.*.x.|
CA 60                                                  |.`|
```

Program **Look1** dumps the stack from the stack pointer just before **Dummy**'s invocation, down to BP after instantiation. Between these two points are **Dummy**'s parameters. **Dummy** has local variables in addition to parameters, however. To view them, you need to follow the stack a little further down, beyond BP to the new position of the stack pointer SP. After **Dummy** has begun executing, SP points to the last word of data in whatever local variables **Dummy** may have.

With some minor modifications, **Look1** shows us the stack from the *before* position of the stack pointer to the *after* position of the stack pointer, including not only all parameters and the return address, but the local variables as well. The modified program is **Look2**. The only change is that SP is taken as the "after" stack marker, rather than BP.

```
PROGRAM Look2;

TYPE
  String30 = String[30];
  IntArray = ARRAY[0..99] OF Integer;

VAR
  OK           : Boolean;
  Register     : Word;
  StackMarker  : Pointer;
  Before,After : Word;
  MyArray      : IntArray;

{$I WRITEHEX.SRC}
{$I VARDUMP.SRC}


FUNCTION Dummy(VAR Values    : IntArray;
                   Fudge     : Integer;
                   Level,Clearance : Char;
                   Message   : String30) : Boolean;

VAR
  Grade,Cutoff : Char;

BEGIN
  INLINE($8B/$C4/$A3/Register);           { Save BP into Register }
  After := Register;
  StackMarker := Ptr(SSeg,Register);      { Make a pointer SS : SP   }
  Writeln('Stack BEFORE result or local variables are modified:');
  VarDump(Output,StackMarker^,Trunc(Before-After));  { Dump stack }
  Dummy := False;                                     { Set function value }
  Grade := 'A';
  Cutoff := 'C';
  Writeln('Stack AFTER result and local variables are modified:');
  VarDump(Output,StackMarker^,Trunc(Before-After));  { Dump stack again }
END;



BEGIN
  INLINE($8B/$C4/$A3/Register);        { Save SP into Register  }
  Before := Register;
  OK := Dummy(MyArray,42,'Q','Z','I was born on a pirate ship.  ');
END.
```

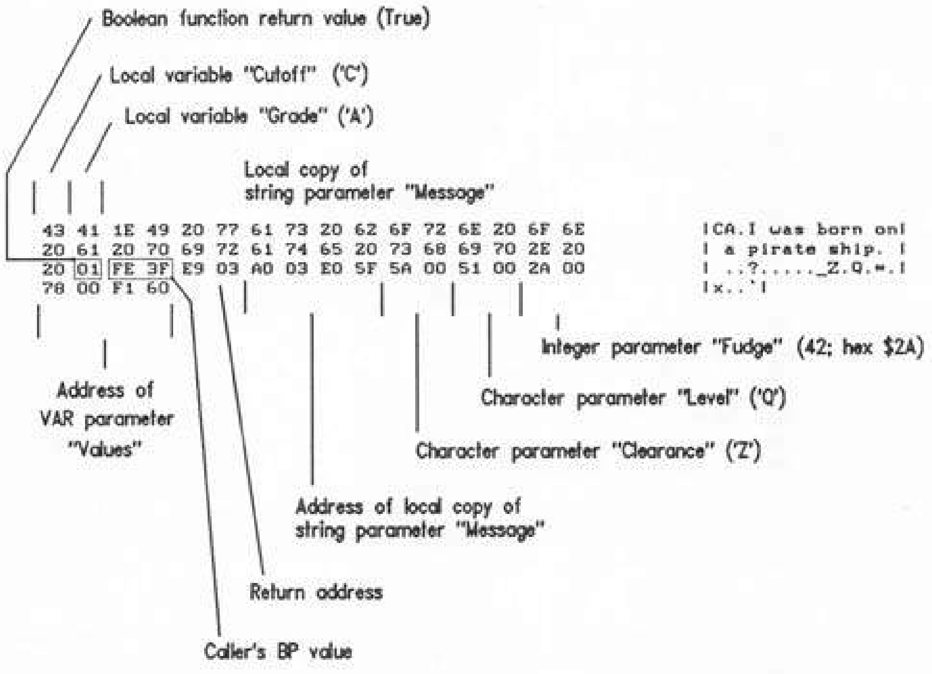
Figure 24.4 shows an annotation of the second of the two dumps of the stack created by **Look2**. The function **Dummy** is the same as that shown in **Look1**. The dump is very similar to the dump produced by **Look1**, except that it goes on a little further into low memory, past the caller's BP value and the return address.

The dump in Figure 24.4 was made *after* the two local variables were assigned values so that the values would appear in the dump.

Figure 24.4

LOOK2's Stack

```
FUNCTION Dummy(VAR Values    : IntArray;
                   Fudge     : Integer;
                   Level,Clearance : Char;
                   Message   : String30) : Boolean;

VAR
  Grade,Cutoff : Char;
```



```
43 41 1E 49 20 77 61 73 20 62 6F 72 6E 20 6F 6E      |CA.I was born on|
20 61 20 70 69 72 61 74 65 20 73 68 69 70 2E 20      | a pirate ship. |
20 01 FE 3F E9 03 A0 03 E0 5F 5A 00 51 00 2A 00      | ..?....._Z.Q.*.|
78 00 F1 60                                          |x..`|
```

Boolean function return value (True)
Local variable "Cutoff" ('C')
Local variable "Grade" ('A')
Local copy of string parameter "Message"
Integer parameter "Fudge" (42; hex $2A)
Character parameter "Level" ('Q')
Character parameter "Clearance" ('Z')
Address of VAR parameter "Values"
Address of local copy of string parameter "Message"
Return address
Caller's BP value

This dump was taken AFTER local variables "Grade" and "Cutoff" were assigned values. Also, the function had been assigned the value True.

Relate the stack dump from **Look2** to the stack diagram in Figure 24.4 for additional clarification of how the stack peek works.

This method of looking at the parameter stack can be a useful first step in designing an external machine language subprogram. Once you decide what parameters will be passed to the subprogram, create a dummy subprogram in Pascal (like **Dummy** in the above example) using those parameters exactly as they will exist in the external subprogram. Pass some easily recognizable values to the subprogram as actual parameters, and then dump the stack from inside the subprogram. Use the hex dump to establish offsets from BP, or to create an assembly language structure for the parameters, as we'll be doing in the section on external assembly language subprograms that occurs a little later on.

Never be too embarrassed to actually *draw* the stack on graph paper to get all the offsets and parameter sizes straight. The bulk of the problems with simple external subprograms is simply not getting the parameters passed correctly. Paper is cheap; time is not. Do whatever you must to get it all straight in your mind, and you will save a great deal of time.

Accessing Local Variables and Parameters from INLINE

Accessing global data items from **INLINE** is best done through register AX, as we described earlier. Turbo Pascal conveniently replaces global variable identifiers with their offsets relative to DS, and a special form of the MOV instruction works handily with AX and offsets from DS.

Accessing subprogram parameters and local variables from within **INLINE** statements is much trickier, since everything has to be specified as an offset from BP. This is *especially* tricky when you want to get at a **VAR** parameter, since what is on the stack at some offset from BP is not the value of the parameter but its 32-bit address.

In this section we'll look at how it's done. As in previous sections, the examples will focus on working with type **Integer** since the bulk of data passed between machine code routines and Pascal code is either 2 bytes or some multiple of 2 bytes in size. The important thing to learn is the general method, so that for each specific case you can puzzle out the details without a great deal of trouble.

Getting at local variables from within **INLINE** is fairly straightforward. Turbo Pascal replaces the names of local variables with their offset from BP, relative to the stack segment register SS. For example, to move the value of AX into a subprogram's local variable, what you need to code up is this:

```
MOV SS:BP+<var>,AX
```

Turbo will obligingly provide the offset from BP represented by **<var>**. But what is the opcode? This is an especially messy instance of the 8086 instruction set coming to the foreground. What we have here is one specific case of the general mnemonic

```
MOV mm,rr
```

meaning, move a 16-bit register into a 16-bit memory location. This mnemonic evaluates to a 2-byte opcode. The first byte, $89, is common to all instances of the generic mnemonic shown above. The second byte is different depending on two things: 1) How we specify mm; in other words, from some register, is the offset given as BP, BP plus a displacement, or one of the many other possible combinations of registers and displacements; and 2) which 8086 register is being moved to mm.

For the example above, the 2-byte opcode is $89/$86. But remember, that's *only* for register AX. The coding for all the other registers is best shown as a table:

```
MOV mm,rr
First opcode                Second opcode
 [BP+dd]        AX    BX    CX    DX    BP    SI    DI    SP
   $89         $86   $9E   $8E   $96   $AE   $B6   $BE   $A6
```

Here, [BP+dd] means, "the word at the address given by register BP plus a 16-bit offset dd." This address is what Turbo Pascal replaces a local variable identifier with.

The following program is a simpleminded example of moving data between machine registers and subprogram local variables:

```
PROGRAM MOVLocal;

USES Crt;

PROCEDURE Foo;

VAR
  Fee,Fie : Integer;

BEGIN
  Fee := 0; Fie := 0;
  INLINE($B8/$11/$00);          {MOV AX,17}
  INLINE($89/$86/Fee);          {MOV Fee,AX}
  Writeln('Fee=',Fee);
  Readln;
  INLINE($8B/$9E/Fee);          {MOV BX,Fee}
  INLINE($89/$9E/Fie);          {MOV Fie,BX}
  Writeln('Fie=',Fie);
  Readln;
END;


BEGIN
  ClrScr;
  Foo;
END.
```

Procedure **Foo** has two local variables, **Fee** and **Fie**. Both these local variables are initialized to zero. An immediate value of 17 is loaded into register AX. Then, the contents of AX are moved into **Fee** by the statement **INLINE($89/$86/Fee);**. This can be verified by displaying **Fee** to the screen. Next, the contents of **Fee** are moved into register BX, and from BX they are moved into local variable **Fie**, which is then displayed to verify that the value of 17 was moved from AX into **Fee**, from **Fee** into BX, and from BX into **Fie**.

Make sure you understand how the second bytes of the MOV opcodes were selected from the table.

Accessing subprogram parameters is done in very much the same way. As with local variable references, when Turbo encounters a value parameter reference within a subprogram, it replaces the reference with the value parameter's offset from BP on the stack. Moving data between registers and value parameters thus requires the identical 8086 opcodes as moving local variables did. Study this program carefully:

```
PROGRAM MOVParameter;

USES Crt;

PROCEDURE Foo(Foe,Fum : Integer);

VAR
  Fee,Fie : Integer;

BEGIN
  Fee := 0; Fie := 0;
  Writeln('Fum=',Fum);
  INLINE($8B/$8E/Foe);              {MOV CX,Foe}
  INLINE($89/$8E/Fee);              {MOV Fee,CX}
  Writeln('Fee=',Fee);
  Readln;
  INLINE($8B/$96/Fee);              {MOV DX,Fee}
  INLINE($89/$96/Fum);              {MOV Fum,DX}
  Writeln('Fum=',Fum);
  Readln;
END;



BEGIN
  ClrScr;
  Foo(42,17);
END.
```

One difference here is that we're using registers CX and DX rather than AX and BX. Notice how that changes the second byte of the opcode, and relate it to the table given earlier.

Program **MOVParameter** takes an integer value from parameter **Foe** and moves it into local variable **Fee** by way of register CX. Then the same value is moved from **Fee** into DX, and from DX into value parameter **Fum**. The important thing to learn in comparing **MOVLocal** and **MOVParameter** is that from **INLINE**'s perspective, subprogram local variables and subprogram value parameters are exactly the same thing, accessed exactly the same way, with the same 8086 instruction.

Now, while it doesn't alter the sense of the opcode, what you should keep in mind in case you have to go troubleshooting your **INLINE** code is that these offsets from BP can be either positive values or negative values. For example, in the little program **MOVLocal** given above, if you add code to examine the value of the offset Turbo replaces the local variable references with, you'll see that the offsets of both **Fee** and **Fie** are negative numbers.

The offsets for value parameters **Foe** and **Fum** in program **MOVParameter**, however, are positive numbers. (For an excellent exercise, code up the **INLINE** statements it would take to examine and display the offset values for **Fee**, **Fie**, **Foe**, and **Fum** in **MOVParameter**.) What goes on here?

Remembering the process of instantiating a subprogram on the stack, think of the position of BP relative to value parameters and local variables on the stack. The stack pointer is copied into BP *after* value parameters are allocated on the stack, but *before* local variables are allocated on the stack (see Figure 24.2). Thus the value parameters must be accessed by adding an offset to BP, while the local variables are accessed by *subtracting* an offset from BP. Adding a negative offset is the same as subtracting an offset. This is why the offsets for **Fee** and **Fie** are negative.

A reminder to beginners: Negative numbers in the 8086 world are encoded in *two's complement* form, such that −1 = $FF; −2 = $FE; −3 = $FD.

Figure 24.5 highlights program **MOVParameter**'s stack as a group of *words* rather than bytes. The dashed lines represent byte boundaries within words. Offsets from BP, however, are always given in bytes.

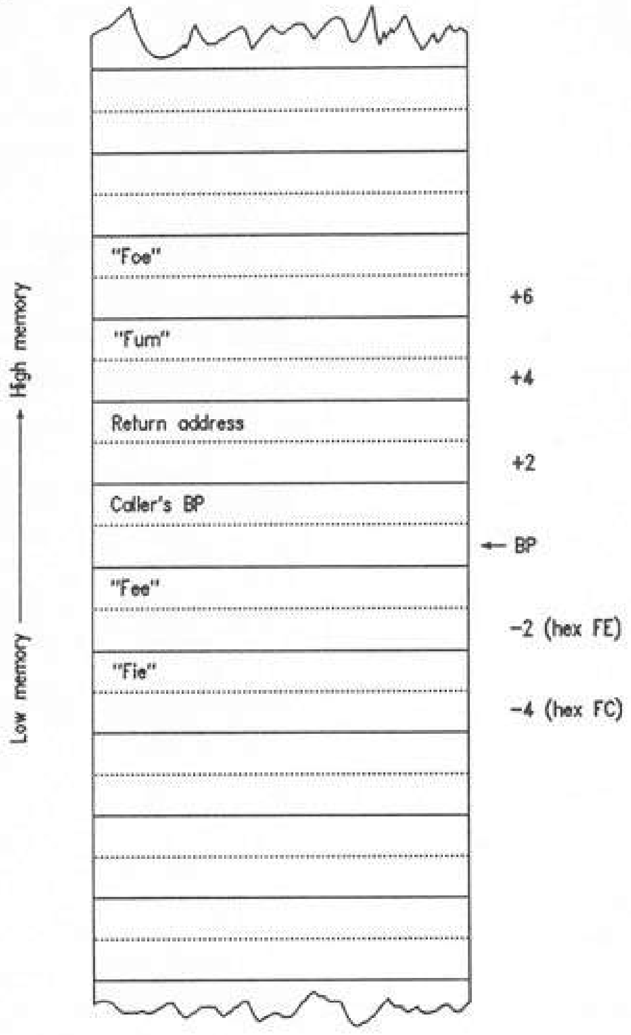
Using VAR Parameters from within INLINE

Compared to using **VAR** parameters, dealing with local variables and value parameters is a snap. Recall that for all **VAR** parameters, and for "oversized" value parameters like arrays, records, and strings, what is pushed onto the stack is not the *value* contained in the actual parameter, but the *address* of the actual parameter or a private copy of it. This introduces another level of indirection in accessing **VAR** parameters: Before we can access the parameter, we have to access its address. Once its address has been moved off the stack and into the right registers, we can work with the parameter itself.

Yes, it's a lot more fooling around than working with value parameters, but the bad news ends there. The good news is that the 8086 contains an instruction designed precisely for taking an address off the stack and putting it in the appropriate register for this kind of indirect addressing.

Figure 24.5

Offsets from BP



NOTE: Negative values are stored in two's complement form in memory and registers

The instruction is LES (Load Pointer Using ES). What it does is pull a 32-bit address from somewhere in memory and put the segment portion of that address into register ES, and the offset portion of the address into the register of your choice. The "somewhere in memory" can be set up to exist at an offset from BP given as a **VAR** parameter identifier within **INLINE**.

Register ES represents what is called the *extra segment*, and it is precisely that: A *spare* segment that can be changed and used for various purposes without disrupting the data segment (DS), code segment (CS) or stack segment (SS). What we're going to do is use the extra segment as a spare data segment, and load the offset from ES into the DI register.

LES is a 2-byte opcode. The first byte is always $C4, and the second byte varies depending on where the address is to be found, and into which register the offset portion of the address is to be loaded. For our purposes in using LES to access **VAR** parameters, the address is found at an offset from BP. Given that, the second byte can be chosen from a simple row of values related to the offset register:

LES coding for accessing VAR parameters:

First byte	Second byte							
	AX	BX	CX	DX	BP	SI	DI	SP
$C4	$86	$9E	$8E	$96	$AE	$B6	$BE	$A6

For our use of DI, the second byte becomes $BE. This little table ought to look familiar; it's the same set of second-byte values used in the register/memory MOV opcode described earlier.

A simple example should give you a feel for how LES is used from **INLINE** to get at a **VAR** parameter:

```
PROGRAM VARParmINLINE;

USES Crt;

VAR
  Flarf : Integer;

PROCEDURE Foo(VAR Fum : Integer);

BEGIN
  INLINE($C4/$BE/Fum);          {LES DI,[BP+<offset>]}
  INLINE($BA/$71/$00);          {MOV DX,$71}
  INLINE($26/$89/$15);          {MOV ES:[DI],DX}
END;


BEGIN
  ClrScr;
  Flarf := 0;
  Foo(Flarf);
  Writeln('Flarf=',Flarf);
  Readln
END.
```

This small program contains a procedure, **Foo**, that has one **VAR** parameter. Procedure **Foo** does only one thing: It places a value of 113 (hex $71) into the **VAR** parameter. If you recall your Pascal, only **VAR** parameters can be changed from subprogram logic, and that is all we're doing in this demo.

Inside procedure **Foo** the first **INLINE** statement takes the address of **VAR** parameter **Fum** off the stack and loads it into registers ES and DI. The second **INLINE** statement loads the immediate value $71 into register DX. The last **INLINE** statement moves the contents of DX into **VAR** parameter **Fum**.

This last **INLINE** statement bears a close look. It's our old friend MOV, but coded up in a somewhat different way. First of all, notice the $26 byte. This is the extra segment prefix. When a segment is not implied (as the stack segment SS is by specifying an offset from BP) the default segment for using MOV is DS. The extra segment prefix tells the 8086 that the segment register for the following instruction is to be ES. Without the prefix, the MOV instruction would have moved data out to the word at DS:DI, whatever that might be.

Try to connect in your mind the use of the ES segment override prefix with access to **VAR** parameters from **INLINE**. It's possible to push the caller's value of DS on the stack and use the LDS pointer load instruction rather than LES, and therefore use DS as the default segment register and not bother with override prefixes, but you must remember to pop the caller's DS value off the stack before the **INLINE** code finishes up. Using ES seems tidier to me, and the fetch overhead incurred for using the prefix is minimal, when balanced against the time required to push and pop DS.

The $89 opcode for MOV is familiar enough; it stands for the MOV mm,rr variant of the MOV instruction. Its second byte is not familiar, however. The second byte depends both on the register operand (in this case, DX) and on the method of addressing the destination memory location. The shorthand for that location is ES:[DI], meaning, "the location pointed to by offset DI from segment register ES." (The square brackets mean we're speaking of the memory location itself rather than its address.) Given those two criteria, a second byte can be chosen from the list of possible second bytes:

`MOV mm,rr coding for accessing VAR parameters at ES:[DI]:`

First byte	Second byte							
	AX	BX	CX	DX	BP	SI	DI	SP
$89	$05	$1D	$0D	$15	$2D	$35	$3D	$25

Since we're using DX in the MOV operation, the second byte (per the table) is $15. Had the operation been MOV mm,BX instead, the second byte would have been $1D.

To recap: Working with a **VAR** parameter first requires that the actual parameter's address be copied from the stack into a segment register and an appropriate offset register. For simple work within **INLINE**, let's standardize on ES for the segment register and DI for the offset register. The instruction LES, when coded correctly for DI and given the offset from BP of the **VAR** parameter's identifier, will take the address from the stack and load it appropriately into ES and DI.

Once the address is in ES:DI, the register/memory instructions can be coded to use the address in ES:DI to access the **VAR** parameter.

Note here that we can do more than MOV values between registers and **VAR** parameters. Any of the following instructions can be used anywhere MOV can be used:

ADC	Add with carry
ADD	Add without carry
AND	Logical AND
CMP	Compare
MOV	Move
OR	Logical OR
SBB	Subtract with borrow
SUB	Subtract without borrow
TEST	Test bit or bits
XCHG	Exchange memory and register
XOR	Logical XOR

I have developed a coding table allowing you to code up the proper opcodes for any of these instructions between any of the 8086 registers and memory, where "memory" means global variables, local variables, value parameters, and **VAR** parameters. I call the table *The Eyeball INLINE Assembler*, and it also includes the most useful variants of *all* other 8086 instructions. I had intended it to be an appendix to this book, but, as it amounts to more than 75 pages all by itself, our publisher rebelled. Your user group may have it on disk, or it is available directly from me at the address on the order sheet.

The program shown below provides a slightly more complicated and considerably more useful example of working with **VAR** parameters from within **INLINE**:

```
PROGRAM FastIncrement;

USES Crt;

TYPE
  IntArray = ARRAY[0..16000] OF Integer;

VAR
  I      : Integer;
  Scores : IntArray;


PROCEDURE Increment(VAR Scores : IntArray;
                        ByHowMuch  : Integer);

BEGIN
```

```
    INLINE($C4/$BE/Scores/        {LES DI,[BP+<offset>]}
           $B9/$80/$3E/           {MOV CX,16000}
           $8B/$9E/ByHowMuch/     {MOV BX,ByHowMuch}
           $26/$01/$1D/           {ADD ES:[DI],BX}
           $47/                   {INC DI}
           $47/                   {INC DI}
           $E2/$FA);              {LOOP -6}

  END;


  BEGIN
    ClrScr;
    { First, zero out the array: }
    FillChar(Scores,SizeOf(Scores),Chr(0));
    FOR I := 0 TO 10 DO
      Writeln(Scores[I]);  { Show first ten values }
    Readln;
    Increment(Scores,72);  { Increment the array }
    FOR I := 0 TO 10 DO
      Writeln(Scores[I]);  { Show first ten values again }
    Readln
  END.
```

Procedure **Increment** is the star of this show, and what it does is add some number (given in **ByHowMuch**) to every element of a 16,000 element integer array. For very large arrays this can be time consuming, and so writing a process like this in machine code can be worthwhile in terms of time saved.

Scores, the array to be incremented, is passed to procedure **Increment** as a **VAR** parameter. Its address is therefore passed on the stack, and the sequence **$C4/$BE/Scores** copies the address from the stack, placing the segment portion in ES and the offset portion in DI. The number of elements in the array (16,000) is loaded as immediate data into register CX. The value parameter **ByHowMuch** is copied from the stack into register BX.

So much for setup; what happens next is an 8086 loop. **Increment**'s real work is done by the ADD instruction coded as the sequence **$26/$01/$1D** (Notice the ES prefix!). The value in register BX is added to the word pointed to by ES:DI. At the outset, this is the first element of the array **Scores**. Now, after the first element of the array has been processed, the DI register is incremented twice. Why? Each element of **Scores** is an integer, and integers are two bytes long. The idea is to point ES:DI to the *next* integer in **Scores**, and DI has to be incremented by two to move past *both* bytes of the integer it is currently pointing to, so it can point to the next integer in line.

The LOOP instruction is a goodie. What it does is decrement the value in CX (which is the number of elements in **Scores**) and then branch *back* by 6 bytes. If you count 6 bytes back from the LOOP opcode you'll find yourself back at the beginning of the ADD opcode, ready to process the next element of **Scores**.

This loop executes again and again, each time decrementing the count in CX. The way LOOP operates, when CX eventually gets to zero, LOOP ceases to branch back, and simply passes control on to the next instruction. In this case, there are no more instructions in the **INLINE** statement, so procedure **Increment** finishes up and returns to the main program.

Notice that −6 is represented in binary fashion as $FA. $FA is the *two's complement* of 6, and in the 8086 world, the negative of a number is given by its *two's complement.* You can calculate two's complements easily by using a programmer's calculator, or the calculator in Borland International's *Sidekick* desk organizer program.

I coded the machine-code portion of **Increment** as a single **INLINE** statement here, as this is standard practice, rather than placing each separate machine instruction in its own **INLINE** statement as I have been doing so far for clarity's sake.

Once you have a pointer to your **VAR** parameter loaded into ES:DI, accessing **VAR** parameters is no more difficult than accessing data in value parameters or subprogram local variables.

Pointer References from INLINE

One other use of the ES:DI addressing mode from **INLINE** involves accessing dynamic variables from within an **INLINE** statement, through pointers. Like a **VAR** parameter, a pointer is a 32-bit address that specifies some location within the 8086's 1 megabyte address space. Also like a **VAR** parameter, a pointer can be loaded into the ES and DI registers, and the pointer's referent (the variable it points to) can be accessed using the exact same opcodes one would use to access a **VAR** parameter. The only difference between working with **VAR** parameters and working with pointers is the way ES and DI are loaded via the LES instruction.

First of all, keep this rather confusing fact in mind: A **VAR** parameter and a pointer passed to a subprogram as a value parameter are identical on a binary basis. After all, a **VAR** parameter *is* a pointer to the actual parameter, even though it is not called a *pointer* in the parameter declaration. Therefore, the *exact same* LES opcodes can be used for both **VAR** parameters and pointers passed as value parameters:

```
INLINE($C4/$BE/Scores         {LES DI,[BP+<offset>]}
       $26/$01/$1D)           {ADD ES:[DI],BX}
```

This example demonstrates this feature. Here, the identifier **Scores** could be *either* a **VAR** parameter, as it was in the little example program **FastIncrement** a few pages back, or a pointer passed to the subprogram containing the **INLINE** statement as a value parameter. In either case, an address has been passed to the subprogram on the stack, under the identifier **Scores**. LES takes this address from the stack and loads the segment portion into ES and the offset portion into DI. The second opcode adds the contents of register BX to whatever is pointed to by the address passed on the stack, whether that address was a **VAR** parameter or a pointer passed as a value parameter.

Pointers, of course, do not have to be passed as parameters. Using a pointer declared as a global variable needs a slightly different LES opcode than the one shown above. A pointer declared as a global variable is simply an address stored in Turbo Pascal's data segment, and it has an offset from the data segment base address in the

DS register. The addressing method is the same used with any global variable, that is, as a 16-bit offset from DS. Assuming a pointer **MyPtr** declared as a global variable, the LES opcode would look like this:

```
INLINE($C4/$3E/MyPtr/            { LES DI,DS:[<offset>] }
       $26/$80/$05/$03);         { ADD WORD PTR ES:[DI],03H }
```

Again, only the second opcode byte is different; LES always has an initial opcode byte of $C4. Turbo Pascal replaces the global identifier **MyPtr** with **MyPtr**'s offset from the DS register, which is what the $C4/$3E opcode requires.

A short program demonstrating pointer access from **INLINE** is given below. A pointer, **MyPtr**, is declared as a variable, and an integer variable is allocated on the heap and given to **MyPtr** as its referent. This dynamic integer variable is given a value of 42 through a normal assignment statement. Then, the **INLINE** statement accesses the dynamic variable pointed to by **MyPtr** and adds 3 to it. The value of **MyPtr**^ is displayed before and after to prove that this shenanigans actually works.

For dynamic variables larger than integers, the offset value in DI may be increased or decreased so that ES:DI points to different parts of the dynamic variable. Keep in mind that on an assembler level within **INLINE**, you can only access 1 byte or 1 word (2 bytes) at a time. To access larger data items you must loop through them, incrementing or decrementing DI after each access.

```
PROGRAM PointerINLINE;

USES Crt;

TYPE
  IntPtr = ^Integer;

VAR
  MyPtr : IntPtr;


BEGIN
  ClrScr;
  New(MyPtr);
  MyPtr^ := 42;
  Writeln('Before: ',MyPtr^);
  INLINE($C4/$3E/MyPtr/            { LES DI,MyPtr }
         $26/$80/$05/$03);         { ADD WORD PTR ES:[DI],03H }
  Writeln('After:  ',MyPtr^);      { '45' should be new value. }
  Readln
END.
```

24.3: A SUMMARY OF MEMORY/REGISTER OPERATIONS FROM INLINE

By now you should be getting the hang of coding up short assembly language sequences via **INLINE**. The real trick, aside from knowing where things are, lies in getting the assembly language mnemonic translated correctly into one or more binary opcodes. Most of the examples in the previous section involved the MOV opcode, since that is one of the simplest and commonest of assembly language operations.

There are a number of other 8086 instructions that can be made to operate on variables and value parameters in a very similar fashion. These operations include addition, subtraction, AND, OR, XOR, exchange, compare, and test. In all of these cases, the opcode you must place in the **INLINE** statement consists of 2 bytes. The first byte specifies the operation, (unless the first byte is a segment override prefix) and the second byte specifies which register is acting or being acted upon. In all cases, a Pascal identifier must be supplied to **INLINE** that specifies which parameter or local variable is acting or being acted upon.

All of these operations can be performed on either 8-bit or 16-bit operands. The opcode bytes are *not* the same in both cases. In all operations, the results of the operation (for example, the sum when you add memory to a register) are placed in the first operand. In other words, for ADD rr,mm the sum is placed in the register.

To assemble an opcode from the table on page 633, first decide which operation you want to perform, say, subtract with borrow (SBB). The opcodes exist in pairs, depending on whether the results of the operation are to be placed in the variable or in the register. Choose the destination for the results and pick one of the two alternative rows. In other words, for subtract with borrow you can have it one of two ways:

```
SBB mm,rr    Difference is placed in the variable (mm)

SBB rr,mm    Difference is placed in the resgiter (rr)
```

Then, decide whether the operation operates on 8-bit or 16-bit operands. Move across to either the 8-bit column or the 16-bit column as appropriate. There's your first opcode byte!

Choosing the second opcode byte is done with the *second byte tables* given on page 634. Your first step here is to zero in on one of the four second-byte tables. Start by deciding the addressing method your operation is to use. In Turbo Pascal **INLINE** terms, this comes down to deciding whether you're working with subprogram value parameters or local variables (which are addressed as offsets from BP); global variables (addressed as offsets from DS); or subprogram **VAR** parameters (addressed through the 32-bit address you must construct in ES:DI). There is one table for global variables, and one table for subprogram **VAR** parameters. But . . . there are *two* tables for subprogram local variables and value parameters.

Subprogram local variables and value parameters are addressed as offsets from BP. In 99.9999999 percent of your cases, that offset will be an 8-bit quantity. (In other words, the offset will be 255 bytes or less.) There is a table for local variables and value parameters where the offset is an 8-bit quantity. For the sake of completeness, I'm including the table to use when you are ranging more than 255 bytes from BP. Now that's a *lot* of stack space to be traversing, and it won't happen often. But when it happens, choose the 16-bit offset table.

OK, you have decided on a second-byte table. The only remaining decision is what register you are using. Find the register that will be involved in your operation, and move down to find your second byte. Finally, if you're using ES as the segment register in accessing a **VAR** parameter, insert the $26 ES segment override prefix *in front of* the opcodes you have just assembled. And that's all there is to it!

Follow this process through these examples until you're quite sure you know what you're doing:

```
SUB Parm8,AH            INLINE($28/$66/Parm8);
XOR SI,Global16         INLINE($33/$36/Global16);
ADC BL,Local8           INLINE($12/$5E/Local8);
OR  DX,Parm16           INLINE($0B/$96/Parm16);
SBB Local16,DI          INLINE($19/$BE/Local16);
CMP Global8,CH          INLINE($38/$2E/Global8);

("Parm" = parameter)
```

First Opcode Bytes for Memory/Register Operations

			First Opcode Byte	
			16-bit mm/rr	*8-bit m/r*
ADC	mm,rr	Add with carry	$11	$10
ADC	rr,mm		$13	$12
ADD	mm,rr	Add	$01	$00
ADD	rr,mm		$03	$02
AND	mm,rr	Logical AND	$21	$20
AND	rr,mm		$23	$22
CMP	mm,rr	Compare	$39	$38
CMP	rr,mm		$3B	$3A
MOV	mm,rr	Move	$89	$88
MOV	rr,mm		$8B	$8A
OR	mm,rr	Logical OR	$09	$08
OR	rr,mm		$0B	$0A
SBB	mm,rr	Sub. w. borrow	$19	$18
SBB	rr,mm		$1B	$1A
SUB	mm,rr	Subtract	$29	$28
SUB	rr,mm		$2B	$2A
TEST	rr,mm	Test bit(s)	$85	$84
XCHG	rr,mm	Exchange	$87	$86
XOR	mm,rr	Logical XOR	$31	$30
XOR	rr,mm		$33	$32

All operations must be coded as
INLINE (<Bytel>/<Byte2>/<identifier>);

For example:
ADD DX,Fum codes as INLINE($03/$96/Fum);

Second Byte Tables

For use with VAR parameters; i.e., where mm is ES:[DI]
(Don't forget the $26 segment override prefix!)

register =	AX	BX	CX	DX	BP	SI	DI	SP
	AL	BL	CL	DL	CH	DH	BH	AH
	$05	$1D	$0D	$15	$2D	$35	$3D	$25

For use with global variables; i.e., where mm is [DS +<offset>]

register =	AX	BX	CX	DX	BP	SI	DI	SP
	AL	BL	CL	DL	CH	DH	BH	AH
	$06	$1E	$0E	$16	$2E	$36	$3E	$26

For use with local variables and value parameters; i.e., where mm is [BP +<offset>]

For use with an 8-bit <offset>:

register =	AX	BX	CX	DX	BP	SI	DI	SP
	AL	BL	CL	DL	CH	DH	BH	AH
	$46	$5E	$4E	$56	$6E	$76	$7E	$66

For use with a 16-bit <offset>:

register =	AX	BX	CX	DX	BP	SI	DI	SP
	AL	BL	CL	DL	CH	DH	BH	AH
	$86	$9E	$8E	$96	$AE	$B6	$BE	$A6

Selecting opcodes for operations that take place between registers or registers and immediate data is much less complex; it is straightforward enough, in fact, to allow the generation of a (large) table containing all the various combinations. This table, which I call the *Eyeball INLINE Assembler*, was too large to include as an appendix to this book but I have given it to many user groups, or you may order it from me directly through the address on the order sheet.

24.4: INLINE MACROS

INLINE statements are inherently messy, and very hard to read unless heavily commented. Turbo Pascal provides a little help in the form of **INLINE** macros, also called **INLINE** directives. At a high level, an **INLINE** macro is simply a procedure or function whose body is a single **INLINE** statement. There is no **BEGIN** or **END**; simply a procedure or function header followed by the **INLINE** statement.

One very simple example is the invocation of the PC BIOS print screen interrupt. You can invoke the print screen interrupt at any point in your program by inserting this **INLINE** statement:

```
INLINE($CD/$05);        { INT 05 }
```

Turning this into a macro only involves giving it a procedure identifier and header:

```
PROCEDURE PrintScreen;

INLINE($CD/$05);
```

Now, instead of having to insert the totally cryptic statement **INLINE($CD/$05)**; to print the screen, you can execute the same code by inserting the identifier **PrintScreen** into your program.

The crucial difference between **INLINE** macros and ordinary procedures and functions is that the code contained by the **INLINE** macro is inserted into your program "in-line." In other words, when the compiler compiles the invocation of **PrintScreen**, it does *not* generate code that calls a procedure somewhere in memory, using the 8086 CALL opcode and its associated RETURN opcode. Instead, the compiler simply places the two binary bytes $CD and $05 into the code stream at that point in the program.

Think of it like the difference between global variables and simple constants. A global variable exists at only one place in the data segment. A simple constant, (numeric or character, like 17, 42, 1.717, J or *) on the other hand, is inserted into the code stream every time it appears in the source code file.

A procedure or function exists at only one place in the code segment, and the same physical series of opcodes is executed each time that procedure or function is called. A separate copy of an **INLINE** macro, on the other hand, is inserted into the code stream each and every time it is invoked in the source code.

Because an **INLINE** macro has no single location in memory, you cannot use the **Addr**, **@**, **Ofs**, or **Seg** functions on **INLINE** macro identifiers.

Parameters

An **INLINE** macro can have parameters, with some restrictions. The parameters are pushed onto the stack. There is no entry or exit code apart from that required to push the parameters onto the stack. Your **INLINE** statement must somehow access the parameters on the stack, and then remove them from the stack before terminating.

The identifiers of the parameters may *not* be referenced by the **INLINE** statement within the macro. Other identifiers (global variables, particularly) may be, however. What you need to do in most cases is to pop byte- or word-sized parameters into registers to use them, or else pop the two words of an address (pointer, **VAR** parameter, or oversized value parameter) off the stack into a register pair like ES:DI, then access the actual parameter through the address.

As an example of the use of a **VAR** parameter with an **INLINE** macro, consider the following:

```
PROCEDURE CapsLock(VAR Target : Char);

INLINE($5F/                 { POP ES }
       $07/                 { POP DI }
       $26/$80/$3D/$61/     { CMP ES:[DI],$61     '  < 'a'? }
       $7C/$0A/             { JL  10 }
       $26/$80/$3D/$7A/     { CMP ES:[DI],$7A     '  > 'z'? }
       $7F/$04/             { JG   4 }
       $26/$80/$25/$5F);    { AND ES:[DI],$5F }
```

Here, because **Target** is a **VAR** parameter, we're passing its address on the stack rather than its actual value. The two words of the address are popped off the stack into registers ES and DI. Next, we test the character at ES:[DI] to see if it is either less than **a** or greater than **z**. If either test is true, we don't need to caps lock the character, because it isn't a lower-case character, but rather an upper case character, digit, symbol, or something else. Finally, the bit-mask $5F is ANDed against the actual parameter. This AND operation forces bit 5 to zero. Since the difference between ASCII lower case and upper case letters lies only in bit 5, forcing bit 5 to zero forces the character to upper case.

This is another good example of calculating jumps in **INLINE**. Count the displacement with the first byte *after* the displacement value as the *0* byte. If the $26 byte following the JG 4 instruction is considered byte 0, counting to 4 puts you in the byte immediately after the $5F value, which is past the end of the macro and exactly where you want to be.

Function Results

INLINE macros are especially handy for writing short functions that return characters, Booleans, numerics, or pointers. To make an **INLINE** macro return a value, you must first declare it as a function, and then in executing the **INLINE** statement, "leave behind" the return value in one of the following places:

1. Single byte values (characters, Booleans, enumerated types) in register AL.
2. Word-sized values (integers, words) in AX.
3. Double word values (pointers) in DX:AX. The segment part must be in DX and the offset part in AX.
4. Type **Real** in DX:BX:AX. The highest word must be in DX, the next word in BX, and the low word in AX.

5. 8087-type values (**Single**, **Double**, **Extended**, and **Comp**) are returned on the top of the 8087 stack. Remember that you need a math coprocessor (or floating point emulation under 5.0) to use the 8087 stack.

The simple numeric function given below takes two value parameters and returns as its result the larger of the two:

```
FUNCTION Larger(I,J : Word) : Word;

INLINE($58/                    { POP AX }
       $5B/                    { POP BX }
       $3B/$C3/                { CMP AX,BX }
       $77/$02/                { JA 2 }
       $8B/$C3);               { MOV AX,BX }
```

This is also a good example of using value parameters with **INLINE** macros. Two POP instructions take the two operands from the stack and put them in AX and BX. The function result is returned in AX, so what we want to do is test the two registers against one another, doing nothing if AX is greater than BX, but moving BX into AX if BX is greater than AX.

The two examples given so far are nothing that you can't do from Turbo Pascal itself, without resorting to the hassle of machine code. Their advantage is that they are *fast.* There is no overhead associated with the CALL and RETURN instructions. If speed is ever a critical issue within some tight loop, you may be able to shave a few cycles here and there by coding up some of the loop's statements as **INLINE** macros. One obvious use would be in coding a hand-optimized graphics primitive like a line draw, where every cycle counts.

There are occasional things that simply can't be easily done from Pascal alone. One of these is returning the instruction pointer (IP) of a particular place in your program. Getting at the instruction pointer is a bit of a trick all by itself. There is no 8086 machine instruction for moving the IP into some other register. The IP is not, in fact, directly accessible at all.

The solution is a trick. What you do is perform a short CALL to the *next* instruction in memory, and then pop the return address off the stack into the register of your choice. This works because CALL does only two things: It pushes the address of the first instruction *after* the CALL instruction onto the stack; and, it transfers control to the location at the offset given in the second byte of the instruction. If this second byte is 0, (which is no offset at all) control is transferred to the next instruction in sequence.

The following **INLINE** macro returns the instruction pointer of the first statement following the statement containing it:

```
{->>>>CodeMark<<<<-------------------------------------------}
{                                                              }
{ Filename : CODEMARK.SRC -- Last Modified 2/14/88           }
{                                                              }
{ This is an INLINE macro that returns a pointer to the code   }
{ that comprises the start of the next statement.  This        }
{ pointer may then be passed to VarDump to display a hexdump   }
{ of that region of code.                                      }
{                                                              }
{                                                              }
{                                                              }
{--------------------------------------------------------------}

FUNCTION CodeMark : Pointer;

INLINE($8C/$CA/        { MOV DX,CS | Put code segment in DX  }
       $E8/$00/$00/    { CALL 0    | Push next address onto stack  }
       $58/            { POP AX    | Pop address offset (IP) into AX  }
       $05/$08/$00);   { ADD AX,7  | Increment AX past assignment code  }
```

The only additional trickery here lies in the ADD AX,7 instruction. This moves the stored IP value past the rest of the **INLINE** macro, so that the value returned to the caller really does reflect the beginning of the next statement, and not a location in the middle of the **INLINE** macro itself.

What is **CodeMark** for? Suppose you are curious to see the opcodes that Turbo Pascal generates for a particular statement or sequence of statements. If you can get the address of the code you wish to examine, you can use the **VarDump** procedure from Section 23.3 to dump any number of bytes beginning at that location:

```
VAR
  CodeSpot : Pointer;

{ Get the IP for the start of the NEXT statement: }
CodeSpot := CodeMark;
I := (I SHR 6) * 54;    { This is the statement to look at }

VarDump(Output,CodeSpot^,20);  { Dump it! }
```

You'll still have to hand-disassemble the bytes into opcodes (an 8086 disassembler is not something you can do in a few bytes, sadly) but if what you're looking for is important enough to dig into the binary code, it's important enough to learn the machine code to find.

CodeMark is unnecessary if you have Turbo Debugger, but Turbo Debugger requires Turbo Pascal 5.0 and will not work with Version 4.0.

24.5: WRITING EXTERNAL MACHINE CODE SUBPROGRAMS

Popular as it is, there are two very good reasons not to do any considerable machine-code work in **INLINE**:

1. It's too much work. Cobbling up a complicated MOV opcode by hand can take the better part of a minute or more. An assembler can generate hundreds of them per second. Also, each time you insert an instruction into the middle of an **INLINE** statement, you must manually recalculate every offset and jump destination. At the risk of sounding too flip, I have to say, hey, man, that's what *computers* are for.
2. There are many more ways to go wrong. (This is both a corollary and a consequence of reason 1.) Calculate an opcode or an offset wrong by a single bit, and your program runs off into the bushes. Assemblers do such things perfectly every time. Assembly language is hard enough to do well without one eye closed and one arm in a sling.

   The way to do it properly is to buy a good MASM-compatible assembler and write your machine code subprograms as Turbo Pascal external routines. Once you do, you'll wonder why you ever bothered with **INLINE**. Throughout this section (throughout this entire book, in fact) when I refer to "the assembler" or "MASM" I am referring to Microsoft's Macro Assembler V5.0, which is current as of this writing (Spring 1988). Borland's own macro assembler, TASM (Turbo ASseMbler) understands MASM syntax completely and can be used interchangeably with MASM, *except* for invocation command syntax. TASM is faster and more powerful, and I recommend it over MASM if you haven't already committed to buying an assembler.

   I recommend creating a separate subdirectory on your hard disk (if you are using one) for MASM. MASM comes with a number of utilities and such that are best not mixed with all your Turbo Pascal work. Gaining easy access to MASM's subdirectory is done by adding the name of the subdirectory to your PATH statement:

```
PATH C:\DOS;C:\TURBO;C:\UTILS;C:\MASM
```

This path statement includes the names of four subdirectories, DOS, TURBO, UTILS, and MASM. Yours may be different, and they may be in any order, just as long as both TURBO and MASM are in the list somewhere.

Linking External .OBJ Files

Turbo Pascal's facility for handling externally-assembled machine-code subprograms is simple and works well. There are two parts to the process. First, you must notify Turbo Pascal that an external .OBJ file exists and is to be linked with the current source file. This is done by using the **$L** compiler directive:

```
{$L GAMER}
```

Here, a file named GAMER.OBJ will be loaded into memory and converted to Turbo Pascal's internal link format. Linking itself is not done just yet. For code to be linked from the .OBJ file, there must be an external function or procedure declaration, which is the second part of the process:

```
PROCEDURE Stick(StickNumber, VAR X,Y : Integer); EXTERNAL;
```

This statement tells Turbo Pascal's "smart linker" to locate the routine called **Stick** in the buffer holding the previously-converted file GAMER.OBJ, and then link the routine into the main body of code being compiled. There is no procedure body given; the **EXTERNAL** reserved word indicates that the procedure body is to be taken from the .OBJ file.

The two statements do not necessarily need to be adjacent in a source code file, but it is good practice to make them adjacent:

```
{$L GAMER}
PROCEDURE Stick(StickNumber, VAR X,Y : Integer); EXTERNAL;
```

In building external machine code subprograms, it is critical to understand exactly what has already been done by the code generator before the first opcode in the external file is executed. As described in Section 24.2, all parameters are first pushed onto the stack. If the external routine is a string function, a 32-bit address of the string work area is also pushed onto the stack, *before* any of the parameters. Finally the return address is pushed onto the stack. The return address may be either 16 bits or 32 bits in size, depending on whether the procedure was declared as NEAR or FAR with the **$F** compiler directive. The caller's BP is not saved on the stack. If you intend to use the stack from within the external routine, you must save the caller's BP and move the stack pointer into BP.

The Absolute Minimum External Assembly Language File

A good way to start explaining how to set up an external assembly language program is to show you the least involved file you can successfully pass through a MASM-compatible assembler. This is it:

```
CODE     SEGMENT BYTE PUBLIC
         ASSUME  CS:CODE
         PUBLIC  GRONK

GRONK    PROC    NEAR
GRONK    ENDP

CODE     ENDS
         END
```

This little snippet will successfully assemble, but no code will be generated, as there are no actual assembly language mnemonics in it. I wrote it entirely in upper case strictly as a typographical convention. It may date me, perhaps, but I am more comfortable reading assembly language in upper case. You can use any combination of upper and lower case that pleases you, because the assembler (like Turbo Pascal) forces all lower case characters to upper case internally before using them.

The symbol **CODE** names a code segment that we are creating. The keyword **BYTE** indicates that the segment is *byte aligned*, which means that the segment is to be assembled such that the code begins at the next available byte address. This may seem obvious, but it is also possible to begin the segment at the next available word, paragraph (16 bytes) or page (256 bytes.) The keyword **PUBLIC** indicates that the segment is to be made available to outside references. The **ASSUME** directive is important, since it tells the assembler that the segment value for certain operations that work on the code segment is to be found in CS. Certain peculiar tricks can be accomplished by making this some other register at some point within the program; for example, by assuming that the code segment value is in SS. Microsoft's *BASICA* gets some of its graphics speed by executing code on the stack, but the thought of doing that makes my hair curl.

Within our segment **CODE** is an assembly language procedure called **GRONK**. To the assembler, **GRONK** is a **PROC** whether it was declared as a function or as a procedure in the Pascal program. In assembly language, there is no syntactic difference between a Pascal function and a Pascal procedure; the difference is all in the way the Pascal program treats register values left behind by the external subprogram.

In this declaration, **GRONK** is a *near* procedure (by the **NEAR** keyword) meaning that it exists within some larger code segment and is located by a 16-bit offset within that code segment. In a Turbo Pascal program, the main program block gets its own code segment, as does each individual unit used by the main program. If you link a near external procedure into the main program, everything is fine. However, if you link a near external procedure into a unit, *that procedure may not be called from outside that unit.* One consequence of declaring **GRONK** as a near proc is that it will expect a 16-bit return address offset on that stack at return time rather than a full 32-bit 8086 machine address.

GRONK could also have been a *far* procedure, by replacing the keyword **NEAR** with the keyword **FAR**. A far procedure may be called from anywhere within the 8086's one megabyte address space. If you are going to create a unit that contains external assembly language procedures or functions, declare them as **FAR**, so that the main program and other units may call them.

Although **GRONK** is the only proc named in this particular file, there can be any number of procs within segment **CODE**, assuming that taken together they contain less than 64K of code. The **GRONK ENDP** directive tells the assembler that the proc is finished. The **CODE ENDS** directive tells the assembler that segment **CODE** is finished. Finally, the **END** directive tells the assembler that the source file is finished.

A Boilerplate External Assembly Language Source File

In Section 24.2, we described what happens when Turbo Pascal instantiates a subprogram. For externals, the compiler will push parameters and the return address onto the stack, but everything from that point on must be done by the external routine itself.

The most important thing here is the saving of the caller's BP on the stack, and the saving of the stack pointer value as the new value of BP. Any external subprogram with parameters *must* perform these steps, and a subprogram without parameters probably won't be especially useful.

I have set up a boilerplate external assembly language source file that builds this necessary BP management into our minimal source code file, and adds a comment header. I cannot overemphasize the importance of commenting assembly language files. *One comment per line* is what is often called *the IBM standard for commenting code.* One comment per line, as far as I'm concerned, is the *minimum* safe level of commenting. Comment it until your friends can read it; then you're probably safe.

```
;==========================================================================
;
;    B O I L E R  -  Boilerplate external assembly language source file
;
;==========================================================================
;
;    by Jeff Duntemann      12 February 1988
;
;
;
;
; BOILER is written to be called from Turbo Pascal V4.0 using the
; {$L}/EXTERNAL procedure convention.
;
; Declare the procedure itself as external using this declaration:
;
; {$L BOILER }
; PROCEDURE Boiler; EXTERNAL;
;
; To reassemble/relink BOILER:
;-------------------------------------
; Assemble this file with MASM or TASM; i.e.  "C>TASM BOILER;"
;

CODE    SEGMENT BYTE PUBLIC
        ASSUME  CS:CODE
        PUBLIC  BOILER

BOILER  PROC    NEAR
        PUSH    BP                      ;SAVE PREVIOUS VALUE OF BP ON STACK
        MOV     BP,SP                   ;SP BECOMES NEW VALUE OF BP


        ; THE BODY OF YOUR EVENTUAL PROC GOES HERE...


        MOV     SP,BP                   ; RESTORE PRIOR STACK POINTER & BP
        POP     BP                      ;  IN CONVENTIONAL RETURN
        RET

BOILER  ENDP
CODE    ENDS
        END
```

Turning Source Code into External Machine Code Files

Creating the actual external machine code files is a good deal easier for Turbo Pascal 4.0 and later than it was for earlier versions. Earlier versions required that the assembly language source code file be assembled, then linked, then turned into a .COM file with the DOS utility EXE2BIN. The second two steps are no longer required. The external .ASM source code file only needs to be assembled to an .OBJ file.

This is done by invoking MASM. MASM reads the source code file, performs its magic, and writes out a relocatable object module to disk. This module is actually a DOS file. The file name is the same as the name of your source file, only with an extension of .OBJ.

From the DOS prompt, a successful MASM session looks like this:

```
C:>MASM BOILER;

Microsoft (R) Macro Assembler  Version 4.00
Copyright (C) Microsoft Corp 1981, 1983, 1984, 1985.  All rights
reserved.


  51062 Bytes symbol space free

      0 Warning Errors
      0 Severe  Errors

C:>
```

Note the semicolon after **BOILER**. It tells MASM that you want all the standard defaults for code and listing files. If you omit the semicolon, MASM will ask you a series of questions about the names you wish to give the various output files. While this is not harmful, it is certainly more trouble than it is worth.

A Simple External Machine Code Routine

Now that the mechanics are in place, it's time to look at a useful real world example of a Turbo Pascal external machine code procedure. This first example is fairly simple conceptually, so even if you haven't yet studied 8088 assembly language in great detail you should be able to follow it.

The example is one in which the additional speed of assembly language really is necessary: full-screen text video. Whenever you have screens, you will have a need to clear them. Turbo Pascal's **ClrScr** routine from the **Crt** unit, while quite fast enough, only clears screens to blank (i.e., fills the screen with space characters, ASCII character 32). There are several halftone characters in the high half of the IBM PC's character set (characters 176 through 178) and clearing a text screen to one of those halftone characters can impart a certain classy look to your screens, reminiscent of Ashton-Tate's elegant *Framework* product.

Another deficiency of **ClrScr** is that it only clears the currently visible text screen buffer, at $B000 for monochrome screens, and $B800 for color screens. I have often set up alternate screen buffers on the heap, where I can "build" a complex screen without forcing the user to watch the cursor flying about, putting the whole thing together. Being able to specify the location of the screen buffer to be cleared can be very handy.

The assembly language external routine **Clears** lets you specify the address of the target buffer, and also improves upon **ClrScr** by allowing you to clear the screen with any specified character. It is defined this way:

```
PROCEDURE Clears(Target    : Pointer; ScreenSize : Integer;
                 Attribute : Integer; CharFill   : Byte);
                 EXTERNAL;
```

You pass it **Target**, a pointer to the screen you wish cleared. A pointer to the visible screen is generated easily enough using Turbo Pascal's **Ptr** function; screens on the heap already have pointers linking them to reality. Since a screen as understood by the Pascal program may be a different size at different times, a size parameter, **ScreenSize**, must also be passed to Clears. **ScreenSize** specifies the total size in bytes of the buffer being cleared. Making sure the pointer **Target** points to a buffer that is actually **ScreenSize** bytes long is up to the programmer. If you clear a larger buffer than **Target** points to, you could destroy essential things in memory, which leads to corrupted data or program crashes.

Clears can specify both the attribute of the characters filling the buffer and the characters themselves. In other words, you can fill a buffer with underlined space characters, or high intensity halftone characters if you like. Note well that parameter **Attribute** is declared as a **Word**, and that the actual attribute information must be placed in the *high* eight bits of the integer value. In other words, if you want to use an attribute byte of $07 (for "normal" text) you must pass a value of $0700 in **Attribute**.

```
;==========================================================================
;
;   C L E A R S  -  Screen clear primitive for Turbo Pascal
;
;==========================================================================
;
;    by Jeff Duntemann      12 February 1988
;
;
;
;
; CLEARS is written to be called from Turbo Pascal V4.0 using the
; {$L}/EXTERNAL procedure convention.  It has the advantage over ClrScr in
; that it can clear a screen stored on the heap, and also that a screen can
; be cleared with a character other than space, like the IBM PC's halftone
; characters, for the Framework effect.  An attribute can be written to the
; cleared buffer as well as a clear character.
```

```
;
; To use CLEARS on the visible screen, you must "doctor" a declared
; pointer to point to either the monochrome or graphics text buffer:
;
; VAR
;   VisibleScreen : Pointer;
;
; VisibleScreen := Ptr($B000,0);  { For the monochrome adapter }
; VisibleScreen := Ptr($B800,0);  { For the color graphics adapter }
;
; Declare the procedure itself as external using this declaration:
;
; {$L CLEARS}
; PROCEDURE CLEARS(Target    : Pointer; ScreenSize : Integer;
;                  Attribute : Integer; CharFill   : Byte);
;                  EXTERNAL;
;
;
; Pass CLEARS the attribute code you wish to use in Attribute
; (typically $0700 for "normal" text display) and the character to "clear"
; with in CharFill.  Use 32 or Ord(' ') to fill with blanks, or you may use
; the "halftone" characters (176-178) for Framework style screens.  Keep
; in mind that the attribute code must be in the HIGH byte of the actual
; parameter passed to Attribute.
;
; EXAMPLES:
;
; To clear the visible screen to normal blanks:
;    CLEARS(VisibleScreen,4096,$0700,' ');
;
; To clear a screen on the heap to a halftone screen:
;    CLEARS(NewScreen,4096,$0700,176);
;
; Obviously, you must have declared VisibleScreen and NewScreen and set
; them up so that they both point to either a screen on the heap or the
; visible screen buffer.  If the pointer Target has a value of NIL,
; CLEARS will return to the calling logic without taking any action.
; Good thing, too--if it did, it would blank your interrupt vector table!
;
;
; To reassemble/relink CLEARS:
;-------------------------------------
; Assemble this file with MASM.  "A> MASM CLEARS;"

CODE    SEGMENT BYTE PUBLIC
        ASSUME  CS:CODE
        PUBLIC  CLEARS
;
; This structure maps the stack at entry to CLEARS:
;
ONSTACK STRUC
OLDBP   DW ?     ;CALLER'S BP VALUE SAVED ON STACK
RETADDR DW ?     ;NEAR RETURN ADDRESS
FILLER  DW ?     ;CHARACTER THAT FILLS THE CLEARED BUFFER
ATTRIB  DW ?     ;ATTRIBUTE FOR THE CLEARED BUFFER
BUFSIZE DW ?     ;SIZE OF THE BUFFER TO BE CLEARED, IN BYTES
BUFOFS  DW ?     ;OFFSET OF BUFFER ORIGIN
BUFSEG  DW ?     ;SEGMENT OF BUFFER ORIGIN
```

```
ENDMRK  DB ?    ;DUMMY LABEL TO MARK END OF DATA ON STACK
ONSTACK ENDS
;


CLEARS  PROC    NEAR
        PUSH    BP
        MOV     BP,SP                       ; CALLING CONVENTION
;
;-----------------------------------------------
; FIRST WE TEST FOR BUFFER = NIL...QUIT IF SO
;-----------------------------------------------
;
        CMP     WORD PTR [BP].BUFSEG,0  ; A NIL POINTER IS A SEGMENT AND
        JNE     START                   ; OFFSET BOTH SET TO 0
        CMP     WORD PTR [BP].BUFOFS,0
        JE      BYE

;
;-----------------------------------------------
; PREPARE THE REGISTERS FOR THE STORE WORD OPERATION
;-----------------------------------------------
;
START:  CLD                             ; CLEAR DIRECTION FLAG
        MOV     AX,[BP].ATTRIB          ; LOAD ATTRIBUTE CODE INTO AX
        AND     AX,0FF00H               ; MASK OUT LOW BYTE OF ATTRIBUTE CODE
        MOV     BX,[BP].FILLER          ; LOAD FILLER CODE INTO BX
        AND     BX,0FFH                 ; MASK OUT HIGH BYTE OF FILLER CODE
        OR      AX,BX                   ; AND COMBINE ATTRIB & FILLER INTO AX
        MOV     BX,[BP].BUFOFS          ; SET DI TO TARGET BUFFER OFFSET
        MOV     DI,BX                   ;  BY WAY OF BX
        MOV     BX,[BP].BUFSEG          ; SET ES TO TARGET BUFFER SEGMENT
        MOV     ES,BX                   ;  BY WAY OF BX
;
;-------------------------------------------------------------
; LOOP TO STORE CHARACTER AND ATTRIBUTE INTO BUFFER BY WORD MOVE
;-------------------------------------------------------------
;
        MOV     CX,[BP].BUFSIZE         ; SET UP COUNTER WITH BUFFER SIZE
        REP     STOSW                   ; DO THE STRING STORE
;
;-----------------------------------
; DONE.. CLEAN UP THE STACK AND LEAVE
;-----------------------------------
;
BYE:    MOV     SP,BP                   ; RESTORE PRIOR STACK POINTER & BP
        POP     BP                      ;  IN CONVENTIONAL RETURN
        RET     ENDMRK-RETADDR-2        ; TRASH 10 BYTES FOR PARMS

CLEARS  ENDP
CODE    ENDS
        END
```

As I've said before, the very worst part about writing external machine code routines is passing parameters between Pascal and the external routine. The parameters are on the stack when the external routine takes over; that much, at least, is done for you.

But getting material off the stack takes a little care. As explained earlier, the stack pointer is copied into BP when the external routine begins executing so that an unmoving "base" exists from which to reference parameters on the stack. All parameters have to be "named" by way of an offset from BP. You can do that manually for each parameter, by using an **EQU** (equate) directive:

```
FILLER   EQU      4[BP]
ATTRIB   EQU      6[BP]
BUFSIZE  EQU      8[BP]
BUFOFS   EQU      10[BP]
BUFSEG   EQU      12[BP]
```

This commonly used notation takes the location pointed to by the BP register (that's what the square brackets around BP mean) and add an offset to them. **4[BP]** means "4 higher than the address pointed to by BP."

This system works fine. However, suppose you decide to add another parameter to the list, and insert it between **ATTRIB** and **BUFSIZE**? You would have to manually recalculate all the offsets past **ATTRIB**.

The assembler can do that more reliably than you can. The way to make the assembler do the hard work is to define a *structure* that can be mapped onto the stack. The five equates shown above have been replaced (in **Clears**) with the following assembly language structure:

```
ONSTACK STRUC
OLDBP   DW      ?    ;CALLER'S BP VALUE SAVED ON STACK
RETADDR DW      ?    ;RETURN ADDRESS OFFSET
FILLER  DW      ?    ;CHARACTER THAT FILLS THE CLEARED BUFFER
ATTRIB  DW      ?    ;ATTRIBUTE FOR THE CLEARED BUFFER
BUFSIZE DW      ?    ;SIZE OF BUFFER TO BE CLEARED, IN BYTES
BUFOFS  DW      ?    ;OFFSET OF BUFFER ORIGIN
BUFSEG  DW      ?    ;SEGMENT OF BUFFER ORIGIN
ENDMRK  DB      ?    ;DUMMY LABEL TO MARK END OF DATA ON STACK
ONSTACK ENDS
```

An assembly language structure is very much like a record in Pascal. The structure is named **ONSTACK**, and it includes everything between the directive **ONSTACK STRUC** and **ONSTACK ENDS**, just as a Pascal record includes everything between the reserved words **RECORD** and **END**.

Both records and structures have component parts called fields. All of **ONSTACK**'s fields but **ENDMRK** are 16-bit words (the directive **DW** means "Define Word") but they don't have to be; if **Clears** were declared a far procedure, the return address would be a full 32-bit machine address, and you would have to use the **DD** ("Define Double") directive instead.

You have probably noticed that there are more fields in **ONSTACK** than there were equates in the parameter list given earlier. The fields **OLDBP** and **RETADDR** have been added to the structure, even though, as the sharper among you may have already checked, neither is referenced anywhere within **Clears**. These two new fields are, in fact, placeholders. As you'll remember, when the real work of **Clears** begins, register BP is pointing to the old value of BP, above which is the return address, above which are the parameters. We don't care what we name **OLDBP** and **RETADDR**; we only need them holding their places because of the way we reference the individual fields within structure **ONSTACK**.

Like Pascal, assembly language structures specify individual fields by *dotting*. In other words, to specify the field **PatientAge** in Pascal record **MedHistory**, you would use the notation **MedHistory.PatientAge**. Assembly language isn't quite so easy, unfortunately. In **Clears** we defined a structure called **ONSTACK** but we didn't say *where* it was. Where it is is at the address pointed to by register BP. In assembly language, we can specify the structure itself by its address rather than its name. In this case, that address is the address contained in BP. So specifying a field within **ONSTACK** is done with this notation:

```
MOV CX,[BP].BUFSIZE
```

When the assembler encounters notation like this, it is smart enough to figure out that the symbol **BUFSIZE** is a field within a structure called **ONSTACK**, and it counts the appropriate number of bytes up-memory from [BP] and grabs the parameter from the structure that we have mapped over the stack. Now, if we are to start the structure at [BP], we have to include the two fields **OLDBP** and **RETADDR** to hold the space between the address contained in BP and where the parameters actually begin.

Oddly enough, we don't use the structure's name **ONSTACK** at all. The name only exists to keep the assembler happy; it likes every entity in a program to have a name even if the name is never put to any use within the program.

Once you understand the way parameters on the stack are referenced, understanding the rest of **Clears** is fairly easy. The first little block of code is a safety feature. A Pascal pointer is an address, and the address passed to **Clears** as **BUFOFS** and **BUFSEG** becomes the starting point for a buffer that is going to be cleared to some value, wiping out whatever was previously in that buffer.

In Turbo Pascal, a pointer with the value **Nil** is two words of zeroes. If we treated it as an address, it would be 0000:0000, which is the start of the 8088 interrupt vector table, and we really *really* don't want to fill any part of that with anything. Plainly, we have to test for **Nil** pointers and return control to the calling logic without taking any action when a **Nil** pointer is detected. If the two compares both come up with 0, indicating a **Nil** pointer, the routine branches to stack cleanup and return without doing anything else.

The rest of **Clears** is essentially one operation: A block move of some 16-bit word into the buffer, using the 8088 block move instruction **STOSW** (*Store String Word*). Most of that operation consists of filling registers with the proper values before actually executing the block move itself. The filler character and the attribute byte are combined into one word stored in AX. The segment and offset portions of the pointer indicating the start of the buffer to be cleared are loaded into ES and DI.

Finally, the number of bytes to be moved is stored into the count register CX, and we let 'er rip with **STOSW**. *Zap!* The buffer is filled with whatever was in register AX. After that, we just clean up the stack and go home.

Simple as it is, **Clears** is versatile and very effective. When you clear a screen with it (even a screen 66 lines high, as mine is) the screen clears *instantaneously*. You simply cannot see the progression of the fill from the top to the bottom of the screen.

Such is the power of assembly language.

Reading the Joystick via Assembly Language

One of the more peculiar omissions from Turbo Pascal (considering the kitchen sink-quality of the product otherwise), is joystick interface. This is all the more puzzling because joystick interface is one of those few areas that requires assembly language to work well, even though a crude interface can be cobbled up in Pascal.

A joystick-reading routine is a good second example of external assembly language subroutines, since it incorporates two **VAR** parameters that return information to the calling program. Getting information into and out of **VAR** parameters is much subtler than simply picking value parameters off the stack.

Before we get into the actual assembly language, let's take a look at the IBM PC joystick interface and how it works.

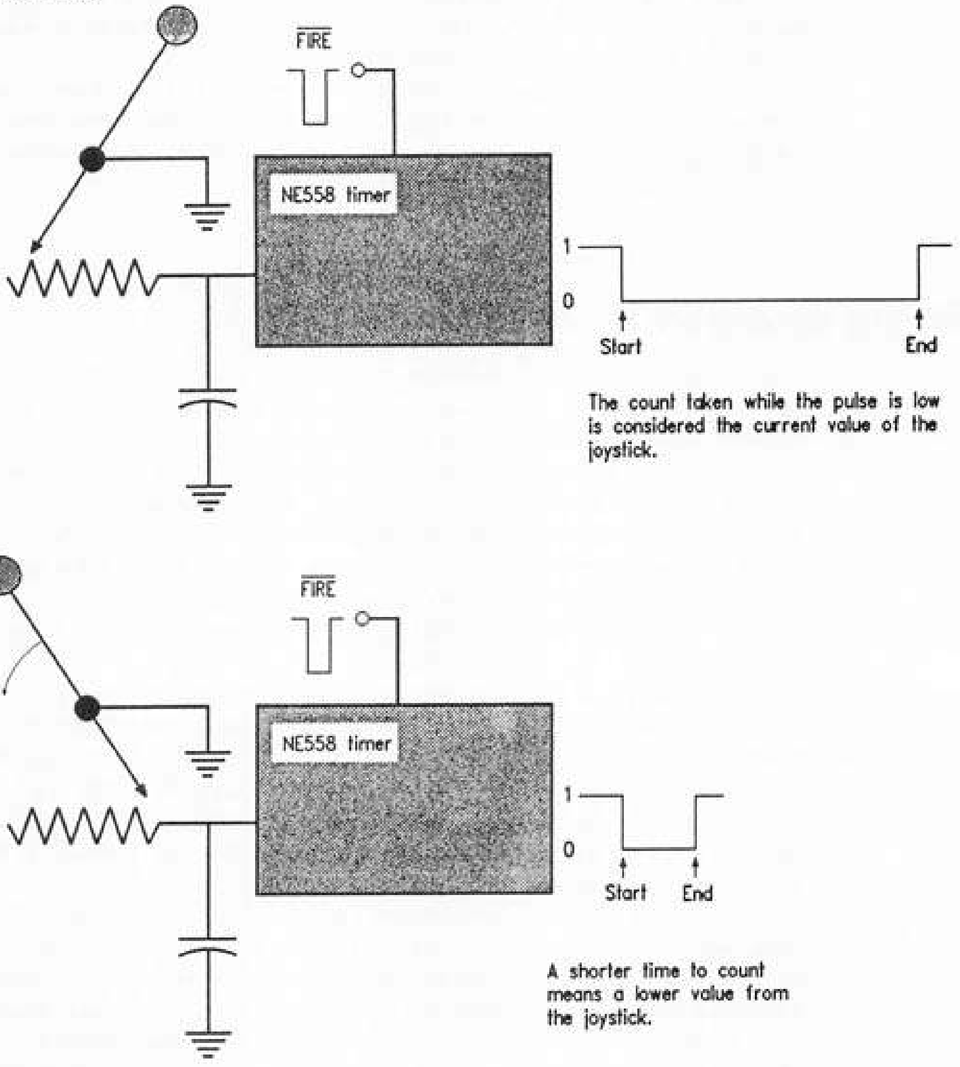
In reading the PC's joystick, we're actually working through a peripheral IBM refers to as the *Game Control Adapter*. This is either a short-slot expansion board or a feature built into a kitchen sink multifunction board. The joystick itself is a pair of potentiometers (variable resistors) mechanically coupled to the stick itself, in such a way that moving the stick in two dimensions runs the two potentiometers up and down their ranges. One potentiometer records the motion in the X axis, and the other records motion in the Y axis.

The game control adapter board itself is actually a generalized means of reading resistance values attached to its inputs. You can actually use the board as a rather crude ohmmeter of limited range and accuracy, and there is nothing special about the joystick mechanism. Any variable resistance of a suitable range can be read through the adapter.

At the core of the game control adapter is a logic element called a *one-shot*. This is a circuit that maintains an output line at a logic one (*high*) until a brief input pulse is applied to its input line. Then the output line goes low and stays low until a resistance-capacitance network *times out;* that is, until the resistance bleeds off the charge stored in the capacitor. The lower the resistance, the more quickly the charge will flow out of the capacitor, and the sooner the output line will return high. If graphed against time, this produces long pulses for high resistance values, and short pulses for low resistance values (see Figure 24.6).

Figure 24.6

One-Shots


FIRE
NE558 timer
1
0
Start
End
The count taken while the pulse is low is considered the current value of the joystick.
FIRE
NE558 timer
1
0
Start
End
A shorter time to count means a lower value from the joystick.

Each joystick requires two one-shot circuits, one for each axis of the joystick. The resistance is supplied by the two potentiometers in the joystick. On the output side, the output lines of the one-shots are coupled to an I/O port with an address of $201.

Software interface is complicated by the fact that each game control adapter supports up to two joysticks, and both joysticks are sampled at once, whether or not any physical joysticks are connected to the adapter.

Sampling works like this: A value is written to I/O port $201. The value doesn't matter; it can by anything at all. The act of writing to the port is what triggers all four one-shots on the adapter at once.

The four output lines of the four one-shots are coupled to bits 0 through 3 of I/O port $201. If you read port $201 without firing the one-shots, all four bits will be set to one. Firing the one-shots forces the 4 bits to 0, and the time it takes for them to return to one is proportional to the resistance of the joystick potentiometers.

The simplest way to measure the time it takes for the bits to change state is to read port $201 repeatedly and test the bit in question until it changes. At each read, a counter is incremented, so that the longer it takes for the bit to change state, the higher the value in the counter. When the bit finally does change state, the counter value is returned to the calling logic as representative of the value of the joystick potentiometer being tested.

The pulses are short enough so that each axis can be tested separately without taking a noticeably long time for the entire sampling operation.

```
;==========================================================================
;
;    S T I C K  -  Assembly language joystick input for Turbo Pascal
;
;==========================================================================
;
;    by Jeff Duntemann     12 February 1988
;      with thanks to Ted Mirecki for additional insights
;
;
;
;
; STICK is written to be called from Turbo Pascal V4.0 using the
; {$L}/EXTERNAL procedure convention.
;
; Declare the procedure itself as an external using this declaration:
;
;    {$L STICK}
;    PROCEDURE STICK(StickNumber : Integer VAR X,Y : Integer);
;                                                  EXTERNAL;
;
; StickNumber specifies which joystick to read from, and the X and Y
; parameters return integers proportional to the joystick's position
; at the moment the stick is sampled.  These integers will vary from
; stick to stick depending on the resistance of the potentiometers
; used within the stick, but will typically from from 3 to 150.
;
; The IBM standard game controller board consists of two pairs of
; one-shots, which output a pulse when triggered by an I/O write to
; I/O port $201.  The length of this pulse is determined by an RC
; time constant circuit the resistance portion of which is the
; potentiometer in the joystick.  As the handle is moved around, the
; two potentiometers (one for X, one for Y) run up and back, changing
; resistance as they go.
;
; To read one of the two joysticks, a dummy value (which may be anything
; at all) is written to I/O port $201.  Port $201 must then be polled
; continuously, incrementing a register at each polling event.  When
; the bit corresponding to that stick's X or Y coordinate changes state,
; the count in the register is returned as that coordinate value at the
; time the stick was sampled.
;
; Here is a map of the joystick bits as returned by port $201:
;
;          |7 6 5 4 3 2 1 0|
;                   | | | |
;                   | | | - - - - - - -> X coordinate, joystick #1
;                   | | - - - - - - - -> Y coordinate, joystick #1
;                   | - - - - - - - - -> X coordinate, joystick #2
;                   - - - - - - - - - -> Y coordinate, joystick #2
;
; One thing to keep in mind is that a bit goes LOW when sampled, and
; you must test for a HIGH on that bit to indicate that the one-shot has
; timed out.
;
;
; To reassemble/relink STICK:
;--------------------------------------
; Assemble this file with MASM.  "C>MASM STICK;"
;
;
; This structure defines the layout of parameters on the stack.
;
ONSTACK   STRUC
OLDBP     DW    ?                ;TOP OF STACK
RETADDR   DD    ?                ;FAR RETURN ADDRESS
YADDR     DD    ?                ;FAR ADDRESS OF X VALUE
XADDR     DD    ?                ;FAR ADDRESS OF Y VALUE
STIK_NO   DW    ?                ;STICK NUMBER
ONSTACK   ENDS

CODE      SEGMENT PUBLIC
          ASSUME  CS:CODE
          PUBLIC  STICK

;  EQUATES FOR ONE-SHOT BITS FOR STICKS 1 & 2

STICK_X   EQU      1
STICK_Y   EQU      2


STICK     PROC    FAR
          PUSH    BP            ;SAVE CALLER'S BP
          MOV     BP,SP         ;STACK POINTER BECOMES NEW BP
          PUSH    DS

;  GET THE X AXIS VALUE FIRST

          MOV     AH,STICK_X          ; MOVE IN THE X TEST BIT
          CMP     [BP].STIK_NO,2      ; SEE IF WE'RE TESTING STICK #1 OR #2
          JNE     TEST_X
          SHL     AH,1          ; SHIFT BIT NUMBERS 2 LEFT FOR STICK #2
          SHL     AH,1
TEST_X:   MOV     AL,1          ; INITIALIZE OUTPUT VALUE
```

```
                MOV     DX,201H         ; SET PORT ADDRESS
                MOV     BX,0            ; AND KEEPING THE RUNUP COUNT IN BX
                MOV     CX,BX           ; LOOP 64K TIMES MAX
                OUT     DX,AL           ; TRIGGER THE ONE-SHOTS
AGAIN_X:        IN      AL,DX           ; READ THE ONE-SHOT BITS
                TEST    AL,AH           ; TEST FOR A HIGH BIT 0
                JE      DELAY           ; WE'RE DONE IF BIT 0 IS HIGH
                INC     BX              ; OTHERWISE INCREMENT BX AND LOOP AGAIN
                LOOP    AGAIN_X
                MOV     BX,-1           ; SET X=-1 IF NO RESPONSE

;  DELAY HERE TO LET THE OTHER THREE PULSES MAX OUT

DELAY:          MOV     CX,512
WAIT:           LOOP    WAIT

;  NOW WE GET THE Y AXIS VALUE

                MOV     AH,STICK_Y          ; MOVE IN THE Y TEST BIT
                CMP     [BP].STIK_NO,2      ; SEE IF WE'RE TESTING STICK #1 OR #2
                JNE     TEST_Y
                SHL     AH,1            ; SHIFT BIT NUMBERS 2 LEFT FOR STICK #2
                SHL     AH,1

TEST_Y:         MOV     SI,0            ; KEEP THE RUNUP COUNT FOR Y IN SI
                MOV     CX,SI           ; SET LOOP LIMIT TO 64K
                OUT     DX,AL           ; FIRE THE ONE-SHOTS AGAIN
AGAIN_Y:        IN      AL,DX           ; READ THE ONE-SHOT BITS
                TEST    AL,AH           ; TEST FOR A HIGH BIT 1
                JE      DONE            ; WE'RE DONE IF BIT 1 IS HIGH
                INC     SI              ; OTHERWISE INCREMENT SI AND LOOP AGAIN
                LOOP    AGAIN_Y
                MOV     SI,-1           ; SET Y=-1 IF NO RESPONSE

;  MOVE RETURN VALUES FROM REGISTERS INTO VAR PARAMETERS X & Y

DONE:           LDS     DI,[BP].XADDR           ;ADDR OF X INTO DS:DI
                MOV     [DI],BX                 ;X VALUE FROM BX TO DS:DI
                LDS     DI,[BP].YADDR           ;DITTO FOR Y VALUE FROM SI
                MOV     [DI],SI

;  IT'S OVER...NOW CLEAN UP THE STACK AND LEAVE

                POP     DS
                MOV     SP,BP                   ; CLEAN UP STACK AND LEAVE
                POP     BP                      ; RESTORE CALLER'S BP

                RET     10

STICK           ENDP
CODE            ENDS
                END
```

Take a look at the assembly language routine **Stick**, which can read either joystick 1 or joystick 2, depending on the value passed in parameter **StickNumber**. **VAR** parameters **X** and **Y** return values proportional to the joystick position in both X and Y axes.

Our friend **ONSTACK** is here, again, with something new: The **DD** directives after fields **XADDR** and **YADDR** mean "Define Double;" they allocate four bytes on the stack for the full 32-bit addresses of the actual parameters plugged into **X** and **Y**.

Before we test anything, we have to determine whether to read joystick 1 or joystick 2. The only difference between testing joysticks 1 and 2 lies in which bits must be watched when reading port $201. Joystick 1 is read on bits 0 and 1, and joystick 2 is read on bits 2 and 3. The equates **STICK_X EQU 1** and **STICK_Y EQU 2** are for joystick 1, and represent bit *values*, not bit *numbers*. To be used in reading joystick 2, both these values must be shifted two bits to the left.

At label **TEST_X,** a dummy value is loaded into register AL, and output to port $201. As mentioned above, this value is not significant; it could be absolutely anything. The important work is done by the address circuitry on the game control board; anytime *anything* is sent to port $201, the one-shot timeout process begins.

The runup count is kept in BX, which is initialized to zero. A maximum runup of 65,536 will be used to prevent the system from hanging in case the game control board has failed electrically and the one-shots never time out. This is done by loading another 0 into CX; the LOOP opcode decrements CX each time it executes, and will "fall through" if CX becomes zero *after* a LOOP opcode. This is why the loop doesn't end immediately, even though we have set CX to 0; LOOP has already decremented CX by one by the time it tests CX for zero, and decrementing a 0 register value yields a value of 65,535.

If LOOP ever terminates after 65,536 iterations, it loads a value of −1 into the return register to indicate to the calling logic that the joystick read has failed.

After outputting the dummy value to port $201 at line 103, a loop is entered that continually reads the one-shots and tests for a high bit. If no high bit is detected, BX is incremented and the loop repeats. Eventually a high bit will be detected, and the count in BX will be proportional to the time it took for the bit to go high.

The X value is returned in BX. The Y value is tested in precisely the same way, except that the runup count is kept in SI rather than in BX.

Perhaps the most important and nonobvious technique demonstrated by **STICK** is the way values are returned from the assembly language subprogram to the pair of Pascal **VAR** parameters X and Y. From the perspective of the assembly code, X and Y are not values but addresses of two variables in the Pascal universe somewhere. Returning a value in X and Y means using those addresses to store values into the variables to which they point.

The means to this end is the LDS opcode. LDS was designed to take a 32-bit address from some location in memory (in this case, on the stack) and break it into two halves. One half (the segment portion) is put into DS (the *DS* in the mnemonic *LDS*) and the other half (the offset portion) is loaded into the register specified after the LDS mnemonic. In the case of **STICK,** this is the DI register. The pointer thus constructed in those two registers DS:DI is then used to MOV a value from a register into one of the variables whose address was passed to **STICK** on the stack.

There are other ways of getting data into a **VAR** parameter, but using LDS (or LES, identical except in using ES where LDS uses DS) is by far the simplest and best.

24.6: RETURNING VALUES FROM EXTERNAL FUNCTIONS

A function in Pascal is a subprogram that returns a value to an expression. A procedure, by contrast, is by itself a statement, and stands alone without returning a value. In Standard Pascal, functions can return only simple types: Integers, reals, Booleans, characters, ordinal types, pointers, and subranges. Turbo Pascal functions, additionally, can return string values as function results.

Pascal functions *cannot* return arrays, records, files, or (sadly) sets.

As mentioned earlier, there is no difference between functions and procedures at an assembly language level. To return a value, an external machine code subprogram *leaves behind* a value in an appropriate place, and after it returns control to the calling logic, the Turbo Pascal runtime code pulls the value from that appropriate place and plugs it into the expression containing the external function invocation.

The thing that tells the runtime code where to look for the value is the type declaration of the function. Every function must be declared with a type, or a Pascal error is generated. For example, integer values are left behind in the AX register. If you define an external function of type **Integer**, the Turbo Pascal runtime will take whatever value is left behind in AX and replace the function's identifier with that value in the expression being evaluated. A function defined as a pointer type, by contrast, will be expected to leave behind the pointer's segment part in DX and its offset part in AX.

The following summaries explain in detail how to return the permissable Pascal types as function results from external functions:

Integers and Integer Subranges

Integer values must be returned in the AX register. The high-order byte of the integer must be in AH, and the low-order byte must be in AL.

Characters, Booleans, Bytes, Enumerated Types and Their Subranges

These data types are all 8-bit types, and are returned in the AL register. In addition, set AH to zero when putting an 8-bit value in AL.

Pointers

The segment portion of a pointer is returned in DX, and the offset portion is returned in AX. The high-order byte of the segment is returned in DH, the low-order byte in DL, and the offset is returned similarly in AH and AL.

Real Numbers

The 6-byte software-only type **Real** is returned in three parts, in three general purpose registers. The high-order word is returned in DX; the middle-order word in BX, and the low-order word in AX.

IEEE real types **Single**, **Double**, **Extended**, and **Comp** are returned on the math coprocessor's internal stack, in the top-of-stack register ST(0). This is done with the 8087 FLD instruction. Keep in mind that under Turbo Pascal 4.0, you *mut* have a math coprocessor installed in the machine to use any of the IEEE real number types. With Turbo Pascal 5.0, math coprocessor *emulation* appeared. Under this scheme, when a compiled program begins running the runtime code determines whether or not a math coprocessor is installed. If one is installed, fine, the program will use the coprocessor. If no coprocessor is installed, a diabolically clever scheme allows the program to emulate the math coprocessor using a scheme of software interrupts. There is a speed penalty, of course, but the end result is that no changes in program code need be made to support a math coproccesor.

Because Turbo Pascal 5.0's emulation of the math coprocessor is complete, your IEEE real functions can return a value on the 8087 stack, even if there is no 8087 in the machine. The FLD instruction will function in either case.

One totally arcane note is that the emulation scheme involves self-modifying code, and so floating point emulation *cannot* be used if code is generated for burning into ROM or EPROM.

Strings

Returning a string value from an external function is *much* simpler with Turbo Pascal 4.0 than in earlier versions. It works like this: Before the Turbo Pascal runtime pushes any of the string function's parameters onto the stack, it pushes the 32-bit address of a string work area onto the stack. This work area is the same physical size as the function's declared string type; that is, if the function is declared to be **String**, then the work area is 256 bytes in size (255 bytes for the string data and one byte for the length byte). If you declare the function to be of a derivative string type (like **String[80]**) the work area will be the size of that derivative type. The work area is somewhere in Turbo Pascal's private data space; just where it is is unimportant, since you always have its address on the stack. After the pointer to the work area, it pushes the parameters, followed by the return address and caller's BP value, just as with any function or procedure.

To return a string value as the function's return value, you must copy the return value into this work area.

The best way to do this is to move the work area's 32-bit address off the stack into a segment and an offset register, using either the LDS instruction (for segment register DS) or the LES instruction (for segment register ES.) Then you can access the work area through the reference syntax ES:[<register>] or DS:[<register>] depending on which segment register you use.

A simple example of a string function is given below. STRPLATE.ASM is a boilerplate string function, in that it doesn't do anything especially useful. What it does is provide you with the correct framework that demonstrates how to get a string value into the work area. You can build on STRPLATE once you understand how it works.

```
;==========================================================================
;
;    S T R P L A T E   --  Boilerplate string external function
;
;==========================================================================
;
;     by Jeff Duntemann      19 February 1988
;
;
;
;
; STRPLATE is written to be called from Turbo Pascal V4.0 using the
; EXTERNAL procedure convention.
;
; Declare the procedure itself as external using this declaration:
;
; {$L STRPLATE}
; FUNCTION STRPLATE(Source : String) : String;
;                                      EXTERNAL;
;
; This function doesn't do anything useful; it simply copies the value of
; Source into the function result.
;
; To reassemble/relink STRPLATE:
;------------------------------------
; Assemble this file with MASM.  "C>MASM STRPLATE;"
;
; The following value can be set to anything between 1 and 255.  80 and 255
; are the commonest and most useful values.  It represents the physical
; length of the string, and MUST match the length of the return string value
; as declared in the Pascal program for the external function.

PHYSLEN   EQU  255

;
; This structure defines the layout of parameters on the stack.  There is
; only one parameter, a string value.  Other parameters would replace the
; PARMPTR field or be added adjacent to it.  Keep in mind that parms
; declared in the function header AFTER the string parm would be added to
; ONSTACK in the position ABOVE the PARMPTR field!
;

ONSTACK   STRUC
OLDBP     DW   ?              ;CALLER'S BP VALUE
RETADDR   DW   ?              ;RETURN ADDRESS

PARMPTR   DD   ?              ;POINTER TO THE STRING PARAMETER

FUNCPTR   DD   ?              ;POINTER TO FUNCTION RESULT WORK AREA
ONSTACK   ENDS

```

```

CODE      SEGMENT BYTE PUBLIC
          ASSUME  CS:CODE
          PUBLIC  STRPLATE

STRPLATE  PROC    NEAR            ;MAKE IT A FAR PROC IF IT'S IN A UNIT!
          PUSH    BP              ;SAVE PREVIOUS VALUE OF BP ON STACK
          MOV     BP,SP           ;SP BECOMES NEW VALUE OF BP


; The "meat" of your own string function should go between the dashed lines.
; leave the stuff outside the dashed lines, alone, unless you change the
; parameter structure or the physical length of the string passed and
; returned.  An assumption made here is that the physical length of the
; string parameter is the same as the physical length of the return value.
; You can easily violate this assumption...but I would advise against it.
;----------------------------------------------------------------------

; For an example operation, let's copy the parm into the function result:

          PUSH    DS                 ;SAVE CALLER'S DS
          LDS     SI,[BP].PARMPTR ;MOVE PARAMETER ADDRESS INTO DS:SI
          LES     DI,[BP].FUNCPTR ;MOVE FUNCTION RESULT ADDRESS INTO ES:DI
          MOV     CX,PHYSLEN+1       ;NUMBER OF BYTES TO MOVE
          CLD                        ;SET DIRECTION FLAG FOR AUTO INCREMENT
          REPZ MOVSB                 ;PERFORM THE BLOCK MOVE

;----------------------------------------------------------------------

          POP     DS                 ;RESTORE CALLER'S DS VALUE
          MOV     SP,BP              ;RESTORE CALLER'S STACK POINTER
          POP     BP                 ;RESTORE CALLER'S BASE POINTER
          RET     FUNCPTR-RETADDR-2   ;CLEAN UP STACK AND RETURN

STRPLATE  ENDP
CODE      ENDS
          END
```

The only "work" performed by STRPLATE is moving the string parameter into the function result. In other words, STRPLATE will return whatever string value is passed in the **Source** parameter as the function result.

This, however, presents an interesting example of moving a string from one place to another in memory using the 8086's very fast block move instruction. The string parameter **Source** is not passed directly on the stack. Instead, a 32-bit address is passed, pointing to the location in memory where the actual parameter actually exists. (This is why we call "actual parameters" what we do.) If the actual parameter passed in **Source** is a quoted string constant, then the address will point to the constant's location in the code segment. If the actual parameter is a variable, the variable's address in the data segment or on the heap will be passed.

So we have two addresses on the stack: One of the actual parameter, and the other of the string work area containing the function result. To move the one into the other, we need to set up a number of registers and then perform a block move function.

Getting the address of parameter **Source** off the stack and into a pair of registers is done with the instruction:

```
LDS SI,[BP].PARMPTR
```

This takes the 32-bit address in the **PARMPTR** field of the **ONSTACK** structure (which is located at BP, and that's why we don't name it directly) and puts the segment portion in DS and the offset portion in SI.

Similarly, the instruction:

```
LES DI,[BP].FUNCPTR
```

takes the address of the string work area and loads the segment portion into ES and the offset portion into DI.

At this point we have both addresses correctly ensconced in the proper registers. The source data for the block move will be the parameter at DS:SI, and the destination data will be at ES:DI. We also need to know how many bytes to move, and this figure must be placed into register CX:

```
MOV CX,PHYSLEN+1
```

We add one to the physical length of the string to account for the length byte, which must be moved as well. The block move instruction keeps track of its progress by decrementing CX after each byte of data is moved. When CX goes to 0, the move is complete.

Next we use the somewhat mysterious instruction CLD to clear the direction flag. The direction flag indicates whether we're going to copy data starting at the *high address* ends of the source and destination or the *low address* ends. If the two data areas happen to overlap, you must begin at the high ends and work downward, nonintuitive though that may be. Since we know that the string parameter and the string work area will never overlap, we can start at the beginning of things and work our way upward. This is specified by clearing the direction flag, using the CLD instruction.

Finally we execute the block move instruction itself:

```
REPZ MOVSB
```

The **MOVSB** instruction makes certain assumptions. These are:

1. The source data is pointed to by DS:SI.
2. The destination data is pointed to by ES:DI.
3. The CX register contains the number of bytes to be moved.

The **REPZ** prefix indicates that the move operation is to be repeated until the 0 register is set, and this happens when CX goes to 0.

Assuming everything is set up correctly, the block move happens in one furious blitz of processor activity initiated by the **REPZ MOVSB** instruction. It is the fastest way to move data in the 8086 world.

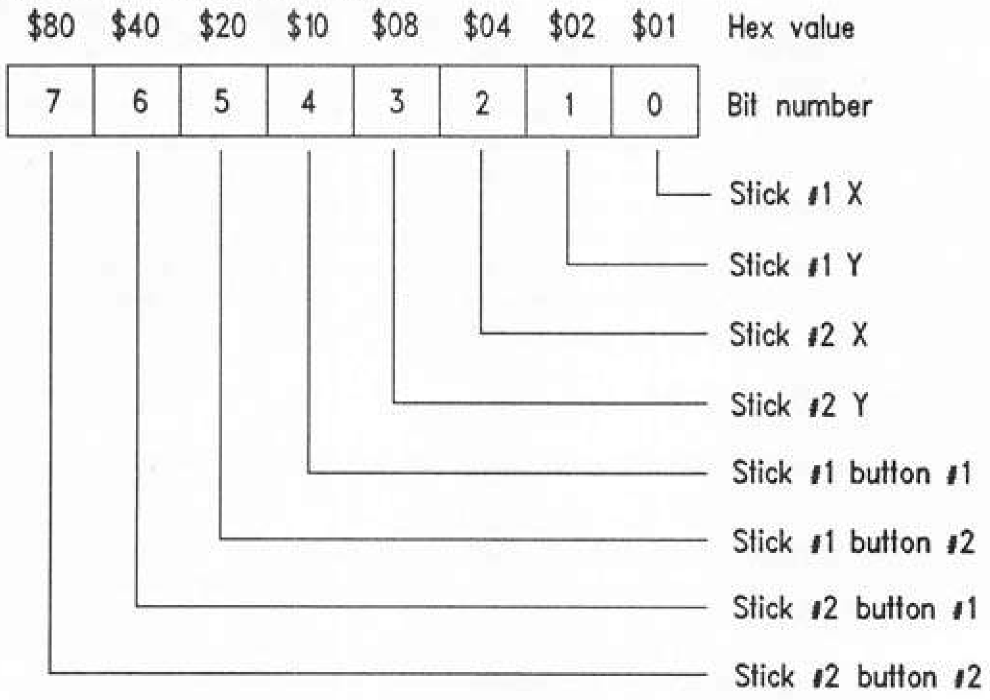
One final note on string functions. When the function is finished and you clean up the stack before returning to the caller, *do not take the address of the string work area off the stack.* In other words, when you move the stack pointer back up the stack with the **RET <n>** instruction, do not move it past the 32-bit address of the work area. In fact, on return, the stack pointer should be pointing *at* the address of the string work area. This is accomplished by the instruction

```
RET FUNCPTR-RETADDR-2
```

Here, we're using the **ONSTACK** structure to calculate the number of bytes by which the stack pointer is to be moved back up the stack, past the parameters. The value we need in this example is 4 (basically, to remove the 4 bytes of the pointer to **Source** from the stack) but by letting the assembler calculate the value, we can put other parameters into **ONSTACK** and not have to recalculate the value attached to the **RET <n>** instruction.

Figure 24.7

Game Controller Bits at Port $201



After the **RET** instruction, the stack pointer will be pointing to the address of the string work area, ready for the Turbo Pascal runtime to copy the contents of the work area into the expression which is called **STRPLATE**.

A Boolean Function to Read the Joystick Buttons

For an example of an external function that returns a Boolean value, consider the problem of returning the state of the switches on a PC joystick. There are two switches on each joystick, and the PC hardware supports two joysticks. The state of all four switches can be read at one time from I/O port $201. If you'll remember from the previous section, the four one-shots that track the X/Y position of the two joysticks are also returned as bit fields in the byte returned from port $201. Bits 0 through 3 are used in determining the joystick positions, and bits 4 through 7 return the current state of the four joystick switches. (See Figure 24.7). The four switch bits are normally high, that is, set to a value of one. When a button is pressed, its bit goes from 1 to 0. The bit remains 0 until the button is released, at which time the bit immediately returns to a value of 1.

I have to be up-front at this point and admit that there is nothing in reading the joystick buttons that can't be done perfectly well from Turbo Pascal. There really isn't any need to read the joystick buttons from an external function. I only provide the external routine because it shows how to return a Boolean value and is both useful and easy to comprehend.

In fact, showing you how to read the buttons in Pascal illustrates an important point often forgotten in designing external machine code routines: The best algorithm for solving a given problem will *not* be the same in both Pascal and assembly language!

```
FUNCTION Button(StickNumber,ButtonNumber : Integer) : Boolean;

VAR
  PortValue : Byte;

BEGIN
  PortValue := Port[$201];    { Read the joystick I/O port }
  IF StickNumber = 1 THEN
    IF ButtonNumber = 1 THEN
      Button := ((PortValue AND $10) = 0)
        ELSE
          IF ButtonNumber = 2 THEN
            Button := ((PortValue AND $20) = 0)
              ELSE Button := False
  ELSE
    IF StickNumber = 2 THEN
      IF ButtonNumber = 1 THEN
        Button := ((PortValue AND $40) = 0)
          ELSE
            IF ButtonNumber = 2 THEN
            Button := ((PortValue AND $80) = 0)
              ELSE Button := False
    ELSE Button := False
END;
```

The Pascal version of **Button** is quite simple. The byte at port $201 is read, and then a somewhat convoluted **IF** statement tests only 1 bit out of the 4 button bits, depending on the values passed to **Button** in parameters **StickNumber** and **ButtonNumber**. The four literal values $10, $20, $40, and $80 represent a mask for the bit to be tested.

You could translate this procedure fairly literally into assembly language, by reading the port into a register, and then testing parameters passed on the stack to 0 in on one out of four tests actually coded.

That, however, is about twice as much code as you need. It's much better to rethink the procedure in terms of 8086 assembly language. While not the tersest implementation, the external function below is certainly terser than hammering a Pascal function into assembler:

```
;==========================================================================
;
;    B U T T O N  -  Function to return the state of the joystick buttons
;
;==========================================================================
;
;     by Jeff Duntemann      12 February 1988
;
;
;
;
; BUTTON is written to be called from Turbo Pascal V4.0 using the
; {$L}/EXTERNAL procedure convention.
;
; Declare the procedure itself as external using this declaration:
;
; {$L BUTTON}
; FUNCTION BUTTON(StickNumber,ButtonNumber : Integer) : Boolean;
;                                                          EXTERNAL;
;
; StickNumber specifies which joystick to read from, and ButtonNumber
; specifies which of the two buttons on that joystick to read.  If the
; specified button is down, BUTTON returns a Boolean value of TRUE.
;
; Yes, this is the long way 'round; assembly language is in no way required
; to read four bits from an ordinary 8088 I/O port.  BUTTON exists only as
; practice in creating assembly language external functions.
;
; The button information is obtained by reading I/O port $201.  The high
; four bits represent the state of the four buttons (two for each of the
; two possible joysticks) at the instant the port is read.  A LOW bit
; represents a button DOWN.  This is why the byte read from the port is
; inverted via NOT before the selected bit is tested.
;
; Here is a map of the button bits as returned by port $201:
;
```

```
;       |7 6 5 4 3 2 1 0|
;         | | | |
;         | | |  - - - - - - - -> Button #1, joystick #1
;         | |  - - - - - - - - -> Button #2, joystick #1
;         |  - - - - - - - - - -> Button #1, joystick #2
;          - - - - - - - - - - -> Button #2, joystick #2
;
; Remember that the return value from this function is passed to the runtime
; code in the AL register.
;
; To reassemble/relink BUTTON:
;-----------------------------------
; Assemble this file with MASM.  "C>MASM BUTTON;"
;
;
; This structure defines the layout of parameters on the stack:
;

ONSTACK   STRUC
OLDBP     DW   ?                ;TOP OF STACK
RETADDR   DD   ?                ;FAR RETURN ADDRESS
BTN_NO    DW   ?                ;BUTTON NUMBER
STIK_NO   DW   ?                ;STICK NUMBER
ONSTACK   ENDS

CODE    SEGMENT BYTE PUBLIC
        ASSUME  CS:CODE
        PUBLIC  BUTTON

BUTTON  PROC    FAR
        PUSH    BP               ;SAVE PREVIOUS VALUE OF BP ON STACK
        MOV     BP,SP            ;SP BECOMES NEW VALUE OF BP

;-------------------------------------------------------------------
; THE BULK OF THIS ROUTINE SETS UP A TEST MASK BY WHICH ONE SINGLE
; BIT OUT OF THE FOUR BUTTON BITS IS TESTED.
;-------------------------------------------------------------------

        MOV     BL,010H          ;START WITH HIGH BIT IN BIT 4
        CMP     [BP].STIK_NO,2   ;ARE WE TESTING FOR JOYSTICK #2?
        JNE     WHICH            ;IF NOT, GO ON TO TEST FOR WHICH BUTTON,
        SHL     BL,1             ; OTHERWISE SHIFT TWO POSITIONS LEFTWARD
        SHL     BL,1             ; SO THAT THE MASK IS ON BIT 6 FOR STICK 2

WHICH:  CMP     [BP].BTN_NO,2    ;ARE WE TESTING FOR BUTTON #2?
        JNE     READEM           ;IF NOT, MASK IS CORRECT; GO READ PORT
        SHL     BL,1             ;OTHERWISE, SHIFT 1 BIT LEFT FOR BUTTON 2

;--------------------------------------------------------------------------
; THE BIT MASK IS NOW CORRECT.  HERE THE BUTTON BITS ARE READ FROM PORT
; $201 AND TESTED AGAINST THE MASK.  NOTE THAT THE BITS AS READ FROM
; THE PORT MUST BE INVERTED SO THAT THE Z FLAG IS SET RATHER THAN CLEARED
; ON AN ACTIVE BUTTON BIT.  (BITS ARE ACTIVE **LOW**, REMEMBER!)
;--------------------------------------------------------------------------

READEM: MOV     DX,0201H         ;SET UP 16-BIT ADDRESS FOR PORT READ
        IN      AL,DX            ;READ BUTTON BITS FROM PORT $201
        NOT     AL               ;MUST INVERT BITS FOR PROPER SENSE
```

```
95                                        ; OF THE Z FLAG AFTER TESTING
96              TEST    AL,BL             ;SEE IF THE DESIRED BIT IS HIGH;
97              JNZ     PUSHED            ;IF SO, BUTTON IS PUSHED;
98              MOV     AL,0              ;ELSE MOVE BOOLEAN FALSE INTO AL
99              JMP     DONE              ;AND GET OUT OF HERE
100
101   PUSHED:   MOV     AL,1              ;BUTTON DOWN; MOVE BOOLEAN TRUE INTO AL
102
103   DONE:     MOV     SP,BP             ;RESTORE PRIOR STACK POINTER & BP
104             POP     BP                ; IN CONVENTIONAL RETURN
105             RET     6
106
107   BUTTON    ENDP
108   CODE      ENDS
109             END
```

The algorithm here is different indeed. Rather than read the port value immediately into a register, the function first tests the passed parameters and sets up a bit mask according to those parameters. Instead of picking from four literals, the code begins with a $10 literal (the bit mask for button 1, switch 1) and shifts it one, two, or three bits to the left depending on the values passed in **ButtonNumber** and **StickNumber**. Note that the Pascal parameter **ButtonNumber** is named **BTN_NO** in the assembly language stack structure, and that **StickNumber** becomes **STIK_NO**. Again, there is nothing in the .OBJ file loaded at compile time to tell Turbo Pascal what you named the stack fields in your assembly language source file. The names do not have to relate to one another at all. For the sake of sense and readability, however, they should!

Only when the mask is properly shifted is the port read and the single test actually made. Note the NOT instruction at line 94. This inverts the byte read from port $201, so that 1s become 0s and vice versa. Recall that when a button is pressed, its bit goes to *zero*. The TEST opcode needs to test for high (1) bits in order to set the 0 flag to the value expected by the Turbo Pascal runtime. TEST would detect the bit change even without the NOT instruction, but in that case the Boolean value returned by **Button** would become **False** when a button were pressed, and remain **True** at all other times. This is counter to what common sense would require in dealing with joystick buttons, so we put the burden of conversion on the assembly language subprogram and not the Pascal code that calls it.

24.7: REPRESENTATION OF TURBO PASCAL DATA ITEMS

Getting Turbo Pascal variables and constants safely into and out of external machine code routines requires understanding how Turbo Pascal represents its data in memory. This section contains a summary of Turbo Pascal's data representation conventions. Keep in mind that a data type as it exists in the data segment may *not* be allocated identically when pushed onto the stack!

Something else to keep in mind about the representation of data types in memory, especially when perusing hex dumps, is that we tend to write hexadecimal numbers down on paper with the least significant byte on the right and move leftward in order of significance, but hex dumps display bytes in increasing significance from left to right. In other words, on paper we would write the hex equivalent of 17,420 as $440C, but in a hex dump we would see the 2 bytes 0C 44.

ShortInts, bytes, characters, and *Booleans* are represented as single bytes in memory when acting as variables in the data area. When pushed on the stack as parameters, they are represented as words, with the high-order byte zero-filled. When written to files they occupy only 1 byte.

Enumerated types are represented the same way bytes, characters, and Booleans are, as no enumerated type may have more than 256 different values. Each constant evaluates to a number, starting with 0. For example, given this enumerated type:

```
TYPE
  Spectrum = (Red,Orange,Yellow,Green,Blue,Indigo,Violet);
```

the constant **Red** evaluates to 0, **Orange** evaluates to 1, and so on. In general terms, a constant from an enumerated type is passed to an external subprogram as the value yielded by Pascal's **Ord** function for that constant.

Integers are represented as words whenever they are used in Turbo Pascal. Bit 7 of the high order byte is the sign bit; a 1 bit in the sign bit indicates a negative quantity.

Pointers are 32-bit quantities, also called *double words* by some. The lower-order word is the segment address, and the higher order word the offset address.

Long Integers (type **LongInt**) are, like pointers, 32-bit quantities. Bit 7 of the highest-order bit is the sign bit; a one bit in the sign bit indicates a negative quantity.

Real numbers come in several distinct and incompatible flavors. Type **Real** is a 6-byte real number format developed by Borland International for Turbo Pascal and used by no one else to my knowledge. The real number types **Single**, **Double**, **Extended**, and **Comp** are implementations of the IEEE standard for real number values.

In Turbo Pascal 4.0, the IEEE types require that a numeric coprocessor (8087, 80287, or 80387) be installed in a machine before a program that uses them will be allowed to run. Turbo Pascal 5.0, on the other hand, has a runtime coprocessor detection scheme that will use the math coprocessor if one is installed in the computer, or emulate the coprocessor in software if no coprocessor is installed. Obviously, programs that must emulate the coprocessor will not run as quickly as those that can use the coprocessor. In fact, the 6-byte type **Real** is faster to use if no coprocessor is available.

Turbo Pascal 6-byte reals keep their exponent in byte 0, and the mantissa in bytes 1 through 5 with the least significant byte of the mantissa in byte 1, and so on.

Real numbers actually have two signs. Bit 7 of byte 5 is the mantissa's sign bit, which indicates whether the real number as a whole is positive or negative.

The other sign is implied in the representation of the exponent in byte 0. This sign indicates to which direction the decimal point moves when converting from scientific notation to decimal notation. You can see this sign by displaying a real number with a value less than 1; a value like .00456 will be displayed on the screen as 4.5600000000E-03. The minus sign after the E indicates that the decimal point must be moved three places to the left when converting this number from scientific to decimal notation.

The value in the exponent is offset by $80. Exponent values greater than $80 indicate that the decimal point must be moved rightward. Exponent values less than $80 indicate that the decimal point must be moved leftward.

Values of type **Real** are stored internally as base two logarithms. In other words, to represent the number 4,673.45, Turbo Pascal stores the exponent to which the number 2 must be raised to yield 4,673.45. This makes multiplying and dividing real numbers much easier on the compiler, since numbers may be multiplied by adding their logarithms, and divided by subtracting their logarithms.

The IEEE real number types are complex and difficult to understand at a binary level. Unless you really and truly know what you're doing, you should *not* try to interpret and manipulate them beneath the level of Turbo Pascal.

The Turbo Pascal runtime support for IEEE real numbers is hard to beat, (especially with Turbo Pascal 5.0) so such tinkering is rarely required. Accordingly, I will not attempt to describe their internal formats here. They are well-documented elsewhere, particularly in Richard Startz's book *8087: Applications and Programming for the IBM PC and Other PC's* (Brady Books, 1983). Startz's book is suprisingly readable, considering that it must make sense of such numeric arcana as negative and positive zero and wraparound infinity.

For purposes of allocating space for them in files or on stacks, it is enough to know that type **Single** occupies 4 bytes; types **Double** and **Comp** occupy 8 bytes, and type **Extended** occupies 10 bytes.

Arrays

One-dimensional arrays are easy enough to understand: The base type of the array is allocated **n** times, where **n** is the number of elements in the array. The lowest-order element of the array is allocated at the lowest memory addresses, and higher-order elements are allocated higher in memory.

One example should suffice. Consider an array of sixteen integers containing the numbers from 17 to 32:

```
IntArray = ARRAY[0..15] OF Integer;

VAR
  Scores : IntArray;
  I      : Integer;

FOR I := 0 TO 15 DO Scores[I] := I + 17;
```

When **Scores** is displayed in a hex dump, it will look like this:

```
11 00 12 00 13 00 14 00 15 00 16 00 17 00 18 00  |................|
19 00 1A 00 1B 00 1C 00 1D 00 1E 00 1F 00 20 00  |................|
```

Scores[0] is dumped as the first two bytes, 11 00, **Scores[1]** as the second 2 bytes, and so on.

Multidimensional arrays are not nearly so straightforward, and the more dimensions you have, the less obvious it is how the compiler allocates array elements in memory. There is a general rule; (there has to be; regardless of what you may think amidst a pile of torn hair at 3:00 A.M., compilers have no free will), but it still requires some thought to get it straight.

The rule is this: From a given point in memory, the *rightmost* dimension (from the array declaration) increases first, followed by the second rightmost, and so on. This is best shown as a table. Consider a three-dimensional array of integers:

```
TYPE
  Space = ARRAY[0..2,0..2,0..2] OF Integer;


High Memory     Space[2,2,2]
                Space[2,2,1]
                Space[2,2,0]
                Space[2,1,2]
                Space[2,1,1]
                Space[2,1,0]
                Space[2,0,2]
                Space[2,0,1]
                Space[2,0,0]
                Space[1,2,2]
                Space[1,2,1]
                Space[1,2,0]
                Space[1,1,2]
                Space[1,1,1]
                Space[1,1,0]
                Space[1,0,2]
                Space[1,0,1]
                Space[1,0,0]
                Space[0,2,2]
                Space[0,2,1]
                Space[0,2,0]
                Space[0,1,2]
                Space[0,1,1]
                Space[0,1,0]
                Space[0,0,2]
                Space[0,0,1]
Low Memory      Space[0,0,0]
```

The lowest element in memory is element [0,0,0], followed by [0,0,1], followed by [0,0,2], and so on.

Accessing an individual array element of a one-dimensional array from within an assembly language subprogram is done by multiplying the size of the individual elements by the index number of the element you want, and adding that offset to the offset of the first element in memory, for which you will already have the address. The address of the array as a whole will be the address of the very byte of the very first element.

If speed is an overwhelming consideration (and if it isn't, why are you working in assembler?), try to make the size of the individual elements some power of 2 (2,4,8,16,32,64,128, etc.). If the size of the base type of the array is a power of 2, you can multiply the index of the desired element by the size of the base type by using shift opcodes. Shifting a register 1 bit to the left multiplies that register by 2; shifting it 2 bits to the left mutiplies it by 4, and so on. For example, if your array's base type is 16 bytes in size, you can index to element 29 of the array by placing 29 in a register and shifting the register four bits to the left (2 to the fourth power is 16). The resulting number is the number of bytes offset into the array where you will find element 29. This is a *great* deal faster than using the MUL and IMUL hardware multiply opcodes, which are in fact very wasteful of machine cycles.

One caution here: array elements are not always declared as indexing from zero. The array declaration:

```
Foo : ARRAY[17..19] OF Integer;
```

is perfectly valid, but it only declares three elements. The Pascal runtime code is smart enough to know that it must subtract 17 from any index value to access an element of array **Foo**, but your assembly language routines will have to take that into account somehow. As always, when you enter The Assembly Zone, you leave all vestiges of Pascal's range protection behind you.

Records

The allocation of record types in memory is not in any way subtle. The first field of the array is at the lowest memory address, followed by the second field, and so on. A simple example will make this clear. Consider this record type, and a record constant defined for the type:

```
TYPE
  FooRec = RECORD
             Alive    : Boolean;
             Points   : Integer;
             Password : String10
           END;
```

```
CONST
  Player : FooRec = (Alive    : True;
                     Points   : 42;
                     Password : 'CORIOLIS  ');
```

If a hex dump of **Player** is displayed, it will look like this:

```
01 2A 00 08 43 4F 52 49 4F 4C 49 53 20 20         |.*..CORIOLIS  |
```

As with all hex dumps shown in this book, the lowest memory address dumped is on the left, with addresses increasing toward the right. The first byte in memory is the first field in the record, **Alive**, which is type **Boolean**. The second field, **Points**, is an integer, and thus takes 2 bytes: 2A 00, which in low-byte-first parlance is hexadecimal for 42. After **Points** comes **Password**, with the 0A value representing the logical length of the string (in this case, 10, the same as the physical length).

In the case of variant records, space is allocated in memory for the largest variant instance of any variant field. The assembly language routine must test the tag field to see which variant to use. For free-union variant records, nothing is passed in the record to indicate which variant is currently in force. The assembly language routine must either be told which variant to use, or else assume one and act accordingly.

24.8: PLACING EXTERNAL MACHINE CODE ROUTINES IN UNITS

There are circumstances under which it *doesn't* make sense to place external machine code routines in Turbo Pascal units, but those circumstances are pretty rare. The only one that comes to mind are truly optimized graphics primitives that are called from within tight loops. Because all calls to routines named in the interface part of a unit are long calls, there is additional push and pop work to be done when using such a routine. Loading an external machine code routine directly to the module that uses it will save that little bit of stack overhead, and in graphics, every cycle counts.

But aside from that, the sensible place for external routines is in a unit.

A Unit for Joystick Interface

A little earlier in this section we described two external assembly language routines, STICK.ASM and BUTTON.ASM, that provide an interface to the standard PC game adapter. There is nothing so speed critical about game board interface as there is about graphics, so we might as well go that last step and describe how to encapsulate those two externals in a Turbo Pascal unit.

There are two ways to go about it. One is to leave both routines as separate asembly language files, and the other is to combine them into a single assembly language source file. We'll discuss both methods here.

In either case, there must be a unit file written in Turbo Pascal to act as a framework for the externals. A simple unit for BUTTON and STICK is shown below:

```
UNIT GameBord;

INTERFACE

FUNCTION  Button(StickNumber,ButtonNumber : Integer) : Boolean;

PROCEDURE Stick(StickNumber : Integer;
                VAR X       : INTEGER;
                VAR Y       : INTEGER);

IMPLEMENTATION

{$L BUTTON}
FUNCTION Button; EXTERNAL;
{$L STICK}
PROCEDURE Stick; EXTERNAL;

END.
```

As you can see, there's nothing much to it from a Pascal standpoint. There is, in fact, no Pascal code of *any* kind in this unit. Every executable instruction is part of one of the two assembly language external routines linked into the unit with the **$L** compiler directive.

I've said it before, but it bears repeating: Both STICK and BUTTON must be declared as FAR procedures within their respective assembly language source code files. All subprograms named in the interface section of a unit are FAR subprograms, whether they are written in Pascal or in assembly language. If you assemble STICK and/or BUTTON as NEAR procedures, they will link successfully with the Pascal unit when you compile it, but when you attempt to call either procedure, your system may hand you a stack error or worse.

A simple joystick interface demo program that uses the **GameBord** unit is given below:

```
{--------------------------------------------------------------}
{                          GAMETEST                            }
{                                                              }
{         Machine-code joystick support demo program           }
{                                                              }
{                          by Jeff Duntemann                   }
{                          Turbo Pascal V5.00                  }
{                          Last update 8/1/88                  }
{                                                              }
{ Nothing complex here.  This program USES the GAMEBORD unit,  }
{ that contains two assembly-language routines for reading     }
{ either of the two joysticks and their associated buttons.    }
{ The program polls the stick continually and reports the      }
{ status of the buttons and the current coordinates of the     }
{ joystick.  To end the program, press any key.                }
{                                                              }
{                                                              }
{                                                              }
{--------------------------------------------------------------}

PROGRAM GameTest;

USES Crt,Gamebord;

VAR
  X,Y : INTEGER;

BEGIN
  ClrScr;
  X := 0; Y := 0;
  GotoXY(2,9); Writeln('X Axis   Y Axis');
  WHILE NOT KeyPressed DO
    BEGIN
      GotoXY(1,5);
      IF Button(2,1) THEN Writeln('Button 1 pressed!')
        ELSE Writeln('Nothing on 1.....');
      IF Button(2,2) THEN Writeln('Button 2 pressed!')
        ELSE Writeln('Nothing on 2.....');
      Stick(2,X,Y);
      GotoXY(1,10);
      Writeln(X:5,'   ',Y:5);
    END;
END.
```

Multiple Assembly Language Procedures in One File

The short Pascal unit given above is perfectly workable, with the only disadvantage that it requires *three* source code files to put together: STICK.ASM, BUTTON.ASM, and GAMEBORD.PAS. The other way to do it is to combine both assembly language source code files into a single source code file. The unit file is almost identical:

```
{--------------------------------------------------------------}
{                          GAMEBORD                            }
{                                                              }
{              Machine-code joystick support unit              }
{                                                              }
{                               by Jeff Duntemann              }
{                               Turbo Pascal V5.00             }
{                               Last update 8/1/88             }
{                                                              }
{ This is the unit body for a unit that actually consists of   }
{ the two assembly language routines in file GAMER.ASM.        }
{                                                             }
{                                                             }
{                                                             }
{-------------------------------------------------------------}

UNIT GameBord;

INTERFACE

FUNCTION  Button(StickNumber,ButtonNumber : Integer) : Boolean;

PROCEDURE Stick(StickNumber : Integer;
                VAR X       : INTEGER;
                VAR Y       : INTEGER);

IMPLEMENTATION

{$L GAMER}
FUNCTION Button; EXTERNAL;
PROCEDURE Stick; EXTERNAL;

END.
```

The important difference is that there is only *one* **$L** compiler directive, indicating that a single .OBJ file, GAMER.OBJ, is to be linked with the unit file. That one **$L** directive can satisfy any number of external references in the implementation section of the unit, as long as each external reference in the unit has a corresponding assembly language routine in the named .OBJ file.

I have combined STICK.ASM and BUTTON.ASM into a single assembly language file, GAMER.ASM, given below:

```
;==========================================================================
;
;    G A M E R  -  Assembly language joystick support for Turbo Pascal
;
;==========================================================================
;
;    by Jeff Duntemann     12 February 1988
;      with thanks to Ted Mirecki for additional insights
;
;
;
;
;
; GAMER is a single assembly-language source file that contains both
; STICK.ASM and BUTTON.ASM, which are given separately elsewhere in
; COMPLETE TURBO PASCAL, 3E.  The purpose of GAMER is to show how multiple
; assembly language procedures may be combined into a single machine-code
; module to lessen program clutter.
;
; The idea is to create a Turbo Pascal unit incorporating the routines in
; this module.  The unit source file is GAMEBORD.PAS.  The headers of both
; routines must be laid out in the interface section of GAMEBORD.PAS, and
; the .OBJ file containing the routines must be loaded into the
; implementation section of GAMEBORD.PAS using the $L compiler directive:
;
; INTERFACE
;
; FUNCTION  Button(StickNumber,ButtonNumber : Integer) : Boolean;
;
; PROCEDURE Stick(StickNumber : Integer;
;                 VAR X       : INTEGER;
;                 VAR Y       : INTEGER);
;
; IMPLEMENTATION
;
; {$L GAMER}
; FUNCTION Button; EXTERNAL;
; PROCEDURE Stick; EXTERNAL;
;
;
;
; GAMEBORD.PAS is given elsewhere in COMPLETE TURBO PASCAL, 3E
;
;
; To reassemble/relink GAMER:
;-------------------------------------
; Assemble this file with MASM.  "C>MASM GAMER;"
;
;


CODE    SEGMENT BYTE PUBLIC       ; THE SEGMENT IS BYTE-ALIGNED
        ASSUME CS:CODE
        PUBLIC BUTTON,STICK       ; THE TWO ACCESSIBLE PROCS IN THIS MODULE



;==========================================================================
;    B U T T O N  -  Function to return the state of the joystick buttons
;==========================================================================
;
; The full function header follows:
;
; FUNCTION BUTTON(StickNumber,ButtonNumber : Integer) : Boolean;
;
; StickNumber specifies which joystick to read from, and ButtonNumber
; specifies which of the two buttons on that joystick to read.  If the
; specified button is down, BUTTON returns a Boolean value of TRUE.
;
; Yes, this is the long way 'round; assembly language is in no way required
; to read four bits from an ordinary 8088 I/O port.  BUTTON exists only as
; practice in creating assembly language external functions.
;
; The button information is obtained by reading I/O port $201.  The high
; four bits represent the state of the four buttons (two for each of the
; two possible joysticks) at the instant the port is read.  A LOW bit
; represents a button DOWN.  This is why the byte read from the port is
; inverted via NOT before the selected bit is tested.
;
; Here is a map of the button bits as returned by port $201:
```

```
;
;      |7 6 5 4 3 2 1 0|
;        | | | |
;        | | |  - - - - - - - -> Button #1, joystick #1
;        | |  - - - - - - - - -> Button #2, joystick #1
;        |  - - - - - - - - - -> Button #1, joystick #2
;         - - - - - - - - - - -> Button #2, joystick #2
;
; Remember that the return value from this function is passed to the runtime
; code in the AL register.
;
;
; This structure defines the layout of BUTTON's parameters on the stack:
;
ONSTACK1  STRUC
OLDBP     DW    ?                   ;TOP OF STACK
RETADDR   DD    ?                   ;FAR RETURN ADDRESS
BTN_NO    DW    ?                   ;BUTTON NUMBER
STIK_NO   DW    ?                   ;STICK NUMBER
ONSTACK1  ENDS

BUTTON  PROC    FAR                 ;ALL PROCS IN A UNIT ARE FAR PROCS
        PUSH    BP                  ;SAVE PREVIOUS VALUE OF BP ON STACK
        MOV     BP,SP               ;SP BECOMES NEW VALUE OF BP

;-----------------------------------------------------------------
; THE BULK OF THIS ROUTINE SETS UP A TEST MASK BY WHICH ONE SINGLE
; BIT OUT OF THE FOUR BUTTON BITS IS TESTED.
;-----------------------------------------------------------------

        MOV     BL,010H             ;START WITH HIGH BIT IN BIT 4
        CMP     [BP].STIK_NO,2      ;ARE WE TESTING FOR JOYSTICK #2?
        JNE     WHICH               ;IF NOT, GO ON TO TEST FOR WHICH BUTTON,
        SHL     BL,1                ; OTHERWISE SHIFT TWO POSITIONS LEFTWARD
        SHL     BL,1                ; SO THAT THE MASK IS ON BIT 6 FOR STICK 2

WHICH:  CMP     [BP].BTN_NO,2       ;ARE WE TESTING FOR BUTTON #2?
        JNE     READEM              ;IF NOT, MASK IS CORRECT; GO READ PORT
        SHL     BL,1                ;OTHERWISE, SHIFT 1 BIT LEFT FOR BUTTON 2

;---------------------------------------------------------------------
; THE BIT MASK IS NOW CORRECT.  HERE THE BUTTON BITS ARE READ FROM PORT
; $201 AND TESTED AGAINST THE MASK.  NOTE THAT THE BITS AS READ FROM
; THE PORT MUST BE INVERTED SO THAT THE Z FLAG IS SET RATHER THAN CLEARED
; ON AN ACTIVE BUTTON BIT.  (BITS ARE ACTIVE **LOW**, REMEMBER!)
;---------------------------------------------------------------------

READEM: MOV     DX,0201H            ;SET UP 16-BIT ADDRESS FOR PORT READ
        IN      AL,DX               ;READ BUTTON BITS FROM PORT $201
        NOT     AL                  ;MUST INVERT BITS FOR PROPER SENSE
                                    ; OF THE Z FLAG AFTER TESTING
        TEST    AL,BL               ;SEE IF THE DESIRED BIT IS HIGH;
        JNZ     PUSHED              ;IF SO, BUTTON IS PUSHED,
        MOV     AL,0                ;SO MOVE BOOLEAN FALSE INTO AL
        JMP     BDONE               ;AND GET OUT OF HERE

PUSHED: MOV     AL,1                ;BUTTON DOWN; MOVE BOOLEAN TRUE INTO AL

```

```
BDONE:  MOV     SP,BP               ;RESTORE PRIOR STACK POINTER & BP
        POP     BP                  ; IN CONVENTIONAL RETURN
        RET     6

BUTTON  ENDP



;==============================================================================
;    S T I C K  -  Procedure to read either joystick
;==============================================================================
;
; The procedure header follows:
;
;    PROCEDURE STICK(StickNumber : Integer VAR X,Y : Integer);
;
; StickNumber specifies which joystick to read from, and the X and Y
; parameters return integers proportional to the joystick's position
; at the moment the stick is sampled.  These integers will vary from
; stick to stick depending on the resistance of the potentiometers
; used within the stick, but will typically from from 3 to 150.
;
; The IBM standard game controller board consists of two pairs of
; one-shots, which output a pulse when triggered by an I/O write to
; I/O port $201.  The length of this pulse is determined by an RC
; time constant circuit the resistance portion of which is the
; potentiometer in the joystick.  As the handle is moved around, the
; two potentiometers (one for X, one for Y) run up and back, changing
; resistance as they go.
;
; To read one of the two joysticks, a dummy value (which may be anything
; at all) is written to I/O port $201.  Port $201 must then be polled
; continuously, incrementing a register at each polling event.  When
; the bit corresponding to that stick's X or Y coordinate changes state,
; the count in the register is returned as that coordinate value at the
; time the stick was sampled.
;
; Here is a map of the joystick bits as returned by port $201:
;
;      |7 6 5 4 3 2 1 0|
;               | | | |
;               | | | - - - - - - -> X coordinate, joystick #1
;               | | - - - - - - - -> Y coordinate, joystick #1
;               | - - - - - - - - -> X coordinate, joystick #2
;               - - - - - - - - - -> Y coordinate, joystick #2
;
; One thing to keep in mind is that a bit goes LOW when sampled, and
; you must test for a HIGH on that bit to indicate that the one-shot has
; timed out.
;
;
;
; This structure defines STICK's parameters on the stack.
;
ONSTACK2  STRUC
OLDBP2    DW   ?                ;TOP OF STACK
RETADDR2  DD   ?                ;FAR RETURN ADDRESS
YADDR2    DD   ?                ;FAR ADDRESS OF X VALUE
```

```
XADDR2    DD   ?                ;FAR ADDRESS OF Y VALUE
STIK_NO2  DW   ?                ;STICK NUMBER
ONSTACK2  ENDS

;  EQUATES FOR ONE-SHOT BITS FOR STICKS 1 & 2

STICK_X   EQU      1
STICK_Y   EQU      2


STICK     PROC     FAR
          PUSH     BP             ;SAVE CALLER'S BP
          MOV      BP,SP          ;STACK POINTER BECOMES NEW BP
          PUSH     DS

;  GET THE X AXIS VALUE FIRST

          MOV      AH,STICK_X         ; MOVE IN THE X TEST BIT
          CMP      [BP].STIK_NO2,2    ; SEE IF WE'RE TESTING STICK #1 OR #2
          JNE      TEST_X
          SHL      AH,1           ; SHIFT BIT NUMBERS 2 LEFT FOR STICK #2
          SHL      AH,1
TEST_X:   MOV      AL,1           ; INITIALIZE OUTPUT VALUE
          MOV      DX,201H        ; SET PORT ADDRESS
          MOV      BX,0           ; AND KEEPING THE RUNUP COUNT IN BX
          MOV      CX,BX          ; LOOP 64K TIMES MAX
          OUT      DX,AL          ; TRIGGER THE ONE-SHOTS
AGAIN_X:  IN       AL,DX          ; READ THE ONE-SHOT BITS
          TEST     AL,AH          ; TEST FOR A HIGH BIT 0
          JE       DELAY          ; WE'RE DONE IF BIT 0 IS HIGH
          INC      BX             ; OTHERWISE INCREMENT BX AND LOOP AGAIN
          LOOP     AGAIN_X
          MOV      BX,-1          ; SET X=-1 IF NO RESPONSE

;  DELAY HERE TO LET THE OTHER THREE PULSES MAX OUT

DELAY:    MOV      CX,512
WAIT:     LOOP     WAIT

;  NOW WE GET THE Y AXIS VALUE

          MOV      AH,STICK_Y         ; MOVE IN THE Y TEST BIT
          CMP      [BP].STIK_NO2,2    ; SEE IF WE'RE TESTING STICK #1 OR #2
          JNE      TEST_Y
          SHL      AH,1           ; SHIFT BIT NUMBERS 2 LEFT FOR STICK #2
          SHL      AH,1

TEST_Y:   MOV      SI,0           ; KEEP THE RUNUP COUNT FOR Y IN SI
          MOV      CX,SI          ; SET LOOP LIMIT TO 64K
          OUT      DX,AL          ; FIRE THE ONE-SHOTS AGAIN
AGAIN_Y:  IN       AL,DX          ; READ THE ONE-SHOT BITS
          TEST     AL,AH          ; TEST FOR A HIGH BIT 1
          JE       DONE           ; WE'RE DONE IF BIT 1 IS HIGH
          INC      SI             ; OTHERWISE INCREMENT SI AND LOOP AGAIN
          LOOP     AGAIN_Y
          MOV      SI,-1          ; SET Y=-1 IF NO RESPONSE
```

```

;  MOVE RETURN VALUES FROM REGISTERS INTO VAR PARAMETERS X & Y

DONE:     LDS    DI,[BP].XADDR2        ;ADDR OF X INTO DS:DI
          MOV    [DI],BX               ;X VALUE FROM BX TO DS:DI
          LDS    DI,[BP].YADDR2        ;DITTO FOR Y VALUE FROM SI
          MOV    [DI],SI

;  IT'S OVER...NOW CLEAN UP THE STACK AND LEAVE

          POP    DS
          MOV    SP,BP                 ; CLEAN UP STACK AND LEAVE
          POP    BP                    ; RESTORE CALLER'S BP

          RET    10

STICK     ENDP



CODE      ENDS
          END
```

At first scan, this might look like nothing more than appending one file to the other and adding an additional comment header for the combined file. Not so! Several changes had to be made to various labels in both source code files, so that no duplicate labels were present in the larger file. Unlike Turbo Pascal, *MASM versions 5.0 and earlier do not understand local identifiers or labels* (MASM 5.1 has them, but not, I think, in a fashion worthy of the name). Both BUTTON and STICK have a label DONE. In combining them, one of the two DONE labels had to go, hence BUTTON's DONE became BDONE.

The same situation comes up in the two stack structures. In the separate assembly language files, both stack structures are called ONSTACK. To differentiate them, I renamed them ONSTACK1 and ONSTACK2. Sadly, the same was true for several of the individual field identifiers within the structures, so to make them different I took the fields in STICK's stack structure and hung a 2 on the end of each field identifier.

This is a real problem if you need to take a great many short assembly language routines and combine them into a single file. There are only so many ways to say DONE and still have them mean anything. Some people just start at AAA and use different permutations of letters for labels. You can imagine what that does to the readability of a 10,000 line assembly language source code file.

The real solution is to use Turbo Assembler, which *does* support local labels, and supports them fully. Every proc within a source code file can have a label @DONE, as long as each @DONE is separated from the others by at least one nonlocal label (the proc name will do just fine).

Turbo Assembler is very new at this writing, but the more I see of it the more I recommend it.

The Road Goes Ever On . . .

Having worked your way entirely through Part Two, you should now have a pretty good feel for all of what Turbo Pascal can do. However, there's a big difference between knowing what it can do and using what it can do in structured, innovative programs. That takes ongoing study and, most of all, *practice*. The only way to become an expert programmer is to get down and program; often, intensely, and well.

At first you would do well to take some of the simple programs presented here and enhance them. Adding code to save and restore graphics screens between the **Scribble** program and disk storage would be an excellent exercise in **BlockRead** and **BlockWrite**.

In time you'll be designing your own programs. Start with small ones, keep program readability as a necessary value, and build on your experience.

You'll be writing thousand-liners in no time at all.

Part Three

Using Your Compiler

INTRODUCTION

In the same way that Part Two of this book was an instruction manual for the Pascal language (as implemented in Turbo Pascal) Part Three of this book is an instruction manual for the Turbo Pascal compiler program itself. Rather than telling you how to write a Pascal program, Part Three will tell you how to get the most from the Turbo Pascal Environment. This is more important now than it ever was under Turbo Pascal 3.0 or earlier versions. Turbo Pascal 4.0 and 5.0 are tremendously more complex than the simple compiler that burst on the scene in 1983.

This part of the book should not be considered a replacement for the *Turbo Pascal Owner's Handbook.* The idea is to help you understand the most commonly used aspects of a very complex product, and to provide a grounding in the philosophy and vocabulary of Turbo Pascal so that when you need to look something up in the *Owner's Handbook,* you'll feel less like you're blundering around utterly in the dark.

25 The Turbo Pascal Environment

"Give me a lever long enough, and a place to stand, and I will move the Earth."

Archimedes was speaking literally about the power of the lever, but behind his words there is a larger truth about work in general: To get something done, you need a place to work, with access to tools. My radio bench in the garage is set up that way: A large, flat space to lay ailing transmitters down, and a shelf above where my oscilloscope, VTVM, frequency counter, signal generator, and dip meter are within easy reach.

Much of the success of Turbo Pascal is grounded in that truth. For the first time, a compiler vendor assembled the most important tools of software development and put them together in an intuitive fashion so that the various tasks involved in creating software flowed easily from one step to the next.

Turbo Pascal versions 1 through 3 were extremely effective in this way. Turbo Pascal versions 4 and 5, about which this book is written, are nothing short of astonishing. In this section I'll give you an overview of the process of making programs with Turbo Pascal. The emphasis will be on the way it all fits together. In later sections, we'll look at the tools and the processes individually and in more detail.

25.1: THE DEVELOPMENT CYCLE

A compiler like Turbo Pascal produces independent executable machine code files. Once a Turbo Pascal program is compiled to disk, you can take the code file to another machine and run it without Turbo Pascal's assistance. This is in contrast to an interpreter like Microsoft's BASICA, which creates a sort of partnership in your computer's memory: Your program file specifies what must be done, but the BASICA program executes the individual machine instructions. Both must be there, in memory, for your program to execute.

What Turbo Pascal uses as a guide to generating executable machine code files is your *source code*, which is one or more files containing human-readable text. This text is what you consider your program. Turbo Pascal acts as a translator, reading in your human-readable source code files and generating a long sequence of binary 8086 machine instructions. This sequence of instructions can be written directly to memory and then executed, or it can be written to disk as an .EXE (EXEcutable) file, which may be loaded and run by DOS like any other DOS program.

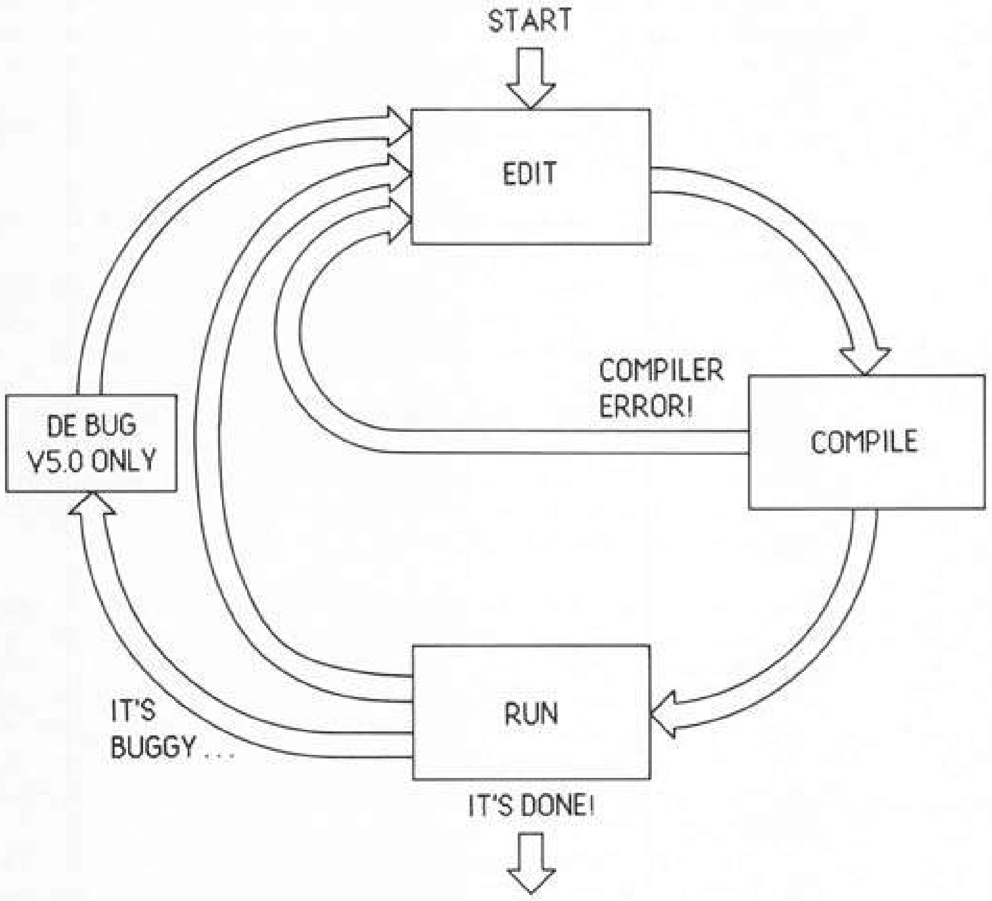
As you know already (or will soon find out) it takes a few tries to get it right. When you first run a Turbo Pascal machine code program file, it will generally malfunction somehow. You then have to look at your source code, try to identify the problem or problems, and make corrections to the source code. Then the whole process begins again: You submit the corrected source code to the Turbo Pascal compiler, and then when the compiler finishes writing out the new machine code program, you must run it and see if anything has improved. At least in the early stages of the game, swatting one bug often generates two or three more. You will probable compile a program dozens or even hundreds of times before you're completely satisfied with it.

If you have any experience with earlier language compilers, that sounds pretty gruesome. It's not so bad, really, because Turbo Pascal's whole *raison d'etre* is to make the process as fast and as painless as possible. When a program compiles in three seconds, suddenly hundreds of compile cycles becomes the work of an afternoon and not a whole week.

Figure 25.1 puts the Turbo Pascal development cycle in diagram form. You start at the top, by using the Turbo Pascal text editor to create a first cut of your program. One keystroke saves the edited source code file to disk. A second keystroke turns the compiler loose, which will read through your source code file and create the executable machine code file.

Figure 25.1

The Turbo Pascal Development Cycle



Bad typing or imperfect understanding of Pascal may cause you to write things in your source code file that the compiler doesn't understand. These lapses in Pascal are called *compiler errors*, but in fact it is *you* who caused them; the poor compiler is simply the bearer of the bad news. When a compiler error is encountered in your source code file, the compiler does its best to explain what is wrong in an error message. Then, it will bring up the text editor again, and put the text cursor as close as it can to where it thinks the error lies.

Your job is to decide what to do, do it, save out the corrected source code file to disk, and compile again. The compiler may find other errors, and if so, you'll find yourself back in the editor, with an error message and a blinking cursor patiently awaiting your corrections.

It make take a few tries, but in time you will produce a source code file that the compiler can fully understand without any compiler errors. The compiler will then dutifully convert the program to executable machine code.

A single keystroke will run the machine code program. It may work perfectly the first time (if you're God, Superman, or General Andrew Jackson) but it probably won't. If it works correctly, you're done! You have a source code file and an executable machine code file. *Back up your source code file.* Don't force victory to be any more expensive than it already is.

However, programs being what they are, one of several things will happen when you run your masterpiece:

1. The machine will go nuts. Or silent. Or nuts and then silent. Reboot. Then think hard about what you're doing in there!
2. You will get a runtime error message. Runtime error messages happen when an otherwise correct program runs into problems as it executes. For example, if your program needs to perform a division, and the divisor happens to be zero, the program will stop with a runtime error.
3. Something reasonable will happen, but it isn't quite what you want. This will be the case most of the time. A variable is displayed that is supposed to contain a name but contains nothing, or it contains half the name, or the wrong name, or the name of something from Mars. Defects like these are called bugs. Get used to them; like the poor, they will always be with you.

If you're using Turbo Pascal 4.0, what you do at this point is study your source code with pencil in hand, ticking off possible trouble spots. Then you go back into the editor and have at it again. If you're using Turbo Pascal 5.0, there are additional tools available to watch what's going on inside your program. These tools are lumped together as *debug support*, and I'll describe them in detail in Chapter 30. They allow you to run your program for awhile, stop and look at the contents of the variables, then start it up and watch some more. They can show you what program statement is executing at any given time. They can be *extraordinarily* helpful.

In any case, you must eventually decide what needs changing in your source code file, then return to the editor and perform the changes.

At that point the cycle begins again.

25.2: YOUR PLACE TO STAND

I suppose the process sounds complicated. It is, but so is driving a car. Do it enough, and you won't even think about half of it. Keep in mind that describing it takes a whole lot longer than actually doing it. On a small program, each loop of the edit/compile/run cycle can take a handful of seconds once you know your way around Turbo Pascal.

As I said before, Turbo Pascal can be seen as a place to stand, with access to tools. The place to stand is what we call the Turbo Pascal Integrated Development Environment, or simply (as I'll call it from now on) the Environment. The Environment is what you see when you execute Turbo Pascal from the DOS prompt:

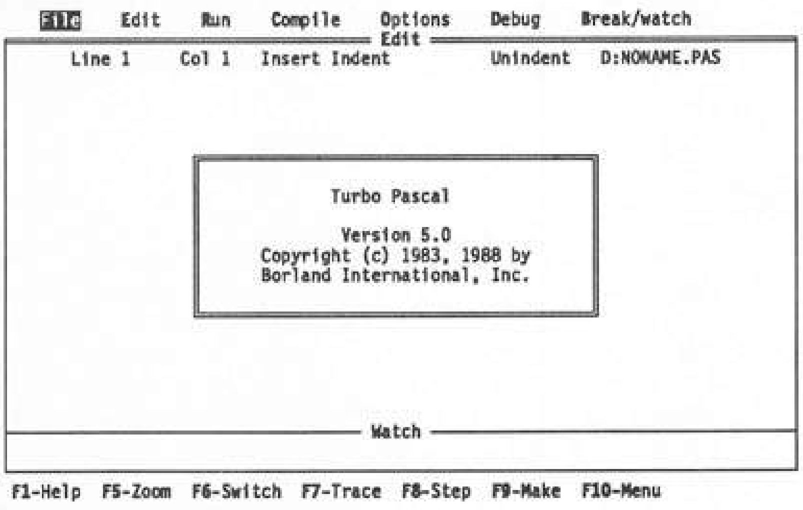
```
C:>TURBO
```

The screen will clear, and what you'll see will look a lot like Figure 25.2. If you're executing Turbo Pascal for the very first time, you'll probably see a title box in the center of the screen, containing the version number of the compiler and Borland's copyright notice. The title box will go away as soon as you press any key; once you set and save any options (as I'll describe later), you won't see the title box again.

The screen as you see it in Figure 25.2 is your view from a height of the Environment, your place to stand. There are four major parts to it (running from top to bottom):

Figure 25.2

The Turbo Pascal Integrated Environment



```
   File   Edit   Run   Compile   Options   Debug   Break/watch
                              Edit
     Line 1     Col 1   Insert Indent        Unindent   D:NONAME.PAS

                         Turbo Pascal

                         Version 5.0
                  Copyright (c) 1983, 1988 by
                  Borland International, Inc.

                              Watch

F1-Help  F5-Zoom  F6-Switch  F7-Trace  F8-Step  F9-Make  F10-Menu
```

1. *The menu bar.* This is the bar running along the top line of the screen. It contains several keywords, each of which represents one or more tools that are available for your use. Most of these keywords contain a "pull-down" menu of tools or further menus.
2. *The edit window.* This is the upper of the two regions into which the screen is divided by a thin line. Within this window you edit your Turbo pascal source code files.
3. *The* Output Window *and the* Watch Window. In Turbo Pascal 4.0, the window beneath the Edit Window is the Output Window. In Turbo Pascal 5.0, the window beneath the Edit Window is the Watch Window. The watch window was introduced with 5.0 and is used to display the values of selected variables while debugging your programs (see Chapter 30). The Output Window is present in Turbo Pascal 5.0, but it must be displayed by pressing Alt-F5. The Output Window is where information written to the screen by your program is displayed.
4. *The prompt bar.* This is the bar running along the bottom line of the screen. It contains information on which single keys represent "short cuts," quick ways of entering frequently used commands.

The Menu Bar

This is "home base." From the menu bar you can work your way down to any of Turbo Pascal's many tools by using the pull-down menus. You can get from *anywhere* within the Environment to the menu bar by pressing the F10 function key. That's a good thing to remember during your first forays out into the Environment: If you get lost, press F10! You'll be back home.

You'll know you're "in" the menu bar when one of its keywords is highlighted in a different color, or in reverse video. The highlighting indicates that that keyword is *selected* and that pressing Return will execute that keyword or bring up its menu.

Once in the menu bar, you can move among the keywords by pressing the right or left arrow keys. Again, pressing Return executes that keyword or brings up its menu. Turbo Pascal 5.0 has more keywords in its menu bar than Turbo Pascal 4.0. I'll explain what each keyword does in detail later on.

The Edit Window

The edit window contains the Turbo Pascal editor. It is a major tool that helps you create and change your source code files. The editor is such a powerful tool that I've devoted all of Section 26 to its detailed use. What I want to do here is simply explain how the edit window relates to the rest of the Environment.

If you're in the menu bar, select the keyword Edit and press Return. The cursor will move into the edit window. Notice that the word Edit at the top of the edit window's line border is highlighted, and furthermore, that the top line of the border is a double line. This indicates that the edit window is the *active* window. The active window is "where the action is," and is the only window that can be *zoomed.*

Zooming a window means expanding that window to occupy the entire screen. The active window can be zoomed by pressing function key F5. A zoomed window has no border, because it is the only visible window on the screen. Once you zoom the active window, you can unzoom it by pressing F5 again. Both windows will reappear.

To my mind, having both windows visible at once is a nuisance, since I like seeing as much of my program as possible when I edit. Furthermore, when all but the simplest programs run they typically use all of the screen, so if you only see the top six lines of a program's output you're almost certainly missing something. There is a way to make Turbo Pascal come up with the edit window in its zoomed state. I'll explain how in connection with the **Z**oom windows item in the **O**ptions menu, in Section 28.2.

The Output Window

The output window is a window into the past, in a sense. When you run a Turbo Pascal program, the Environment "steps aside" for a bit, and the entire screen is devoted to your program's output. *Output* means words and numbers written to the screen by your program, using procedures such as **Write** and **Writeln**. Once your program finishes executing and you return to the Environment, the Output Window will contain the most recent output from your program. This allows you to study your source code in the Edit Window, and relate the statements in the source code to the output in the Output Window.

The fact that the Output Window is smaller than the entire screen means that not all of your output may be visible at once. You can *pan* the Output Window up and down by using the up and down arrow keys. You will see the cursor flashing at the position it held just before you re-entered the Environment.

To zoom the Output Window, you must first make it active. Function key F6 switches the active window from edit to output or from output to edit. Function key F5 zooms the current window, whichever one it is. Try it a few times, switching from one window to the other, and zooming each. It will soon seem natural.

You also have the option, at any time, of zooming to the Output Window with the Alt-F5 shortcut key. This is handy when you just want to "duck out" to see what the current output screen looks like.

As I mentioned above, screens are too small to begin with, and sharing even a 43-line EGA screen is something I don't care for. Single keystrokes allow you to move from the edit window to the Output Window, so there's very little reason to keep both on the screen at once.

The Watch Window (Version 5.0)

A third window was added to Turbo Pascal with release 5.0: The Watch Window. A *watch* is a continuous monitor on the value of a variable, typed constant, or expression. If you set a watch on a given variable, you will be able to watch that variable's value change as you single step your way through a program, or execute portions of the program between breakpoints. At each breakpoint or pause between steps, you will see the current value of the variable in the Watch Window.

Even though there are three available windows in Turbo Pascal 5.0, you will only be able to display two of them at any one time. By default, the Watch Window is displayed and the Output Window is not. You can replace the Watch Window with the Output Window by making the Watch Window active (F6) and then pressing Alt-F6.

The uses of the Watch Window will be explained in detail in Chapter 30.

The Prompt Bar

If you squirm at the thought of memorizing the function keys' functions, relax. The prompt bar was created as a crib sheet for Turbo Pascal's short cut commands. There are two flavors of short cut commands: The function keys and Alt-key combinations. You've seen some of the function keys in action if you've been reading along: F5 zooms and unzooms the current window, for example. I haven't said much about Alt-key commands yet, but there's a very important one to remember: Alt-X will end execution of Turbo Pascal and get you back to DOS.

Ordinarily, the prompt bar contains summaries of the function key commands. However, if you hold down the Alt-key for a moment, the prompt bar will change to a summary of the most important Alt-key commands instead.

The precise meaning of some of the commands changes depending on what you're doing, and the prompt bar changes to reflect the exact meaning of the displayed commands. For example, if the edit window is currently active, the prompt for F6 will be **F6-Output**, indicating that pressing F6 will make the output window active. But once you press F6 and make the Output Window active, the prompt for F6 will immediately change to **F6-Edit**, meaning that pressing F6 again will take you back to the Edit Window.

Pay close attention to the prompt bar while you're first learning your way around the Environment, and you'll rarely get lost.

Menus

The Turbo Pascal Environment is menu-driven. Everything that the Environment can do can be done by working your way through the menu structure to the item you want to select. A menu is a rectangular box with some number of items in it, (each item being a word or phrase) plus a *bounce bar* that you can use to select one item from the menu. Sometimes the menus are *nested;* that is, one item in a menu, when selected, causes another menu to appear over the first, with a lineup of further choices.

Within each menu there are two ways to select one item from those displayed. The most obvious way is to move the bounce bar with the up/down arrow keys so that it highlights the item you want, and then press Return. You may notice in looking at Turbo Pascal's menus that the first character in each item on a menu is displayed in bold. This isn't purely decorative; it's to remind you that pressing the first letter of any menu item will select that item, *without* pressing Return. The wording of the items in each menu has been carefully chosen so that no two items begin with the same letter.

Short Cuts

One criticism of menu-driven systems is that the menus, while invaluable to beginners, tend to get in the way of expert users. The way around this problem is to provide two paths to a command whenever possible: One through the menu system, and another through a single-key *short cut.* (The *Turbo Pascal Owner's Handbook* calls these *hot keys,* but I object to that usage, because I prefer that *hot key* apply *only* to keys that invoke memory resident programs like Sidekick or WindowDOS.)

The best example of this duality involves the command that saves the current edit file to disk. You can bring down the **Files** menu and select the **Save** command: Or, you can simply press the F2 function key from anywhere in the Turbo Pascal Environment. Saving your source file to disk early and often is a crucial part of creating good software with minimal work, and Turbo Pascal makes it as easy as possible to save your source. (Turbo Pascal, in fact, has an option to make saving even easier, by *automatically* saving your source code file to disk every time you run the program or "shell out to DOS", as I'll explain in Section 25.3).

There isn't a short cut for *every* command, but there are short cuts for all the most important and commonly-used commands. These short cuts are summarized in the tables on the following pages. There are a fair number of new shortcuts in Turbo Pascal 5.0, as well as some that have been redefined, so it seems best to give each version a separate table (Tables 25.1 and 25.2).

Backing Out

In general, the Environment's philosophy is to use the Return key to select and execute a command, and the Esc key to "back out" of a command or a menu without completing the command or selecting anything from the menu. If you acidentally enter a menu that you have no need for, press Esc, and you will return to the position from which you selected that menu.

Dialog Boxes and File Selection Matrixes

You'll encounter two other types of critter in your exploration of the Turbo Pascal Environment. One is the *dialog box.* This is a rectangular window that pops up in order to request some keyboard input from you. Turbo Pascal asks, and you answer, hence the "dialog." Typically, the dialog box will have a title in the top line of the box's border, and the title will suggest the required input.

Table 25.1
Turbo Pascal 4.0 Shortcuts

Shortcut Key	*Meaning*
F1	Invokes the Turbo Pascal help system.
F2	Saves the file currently being edited.
F3	Lets you load a new file into the editor.
F5	Zooms and unzooms the currently active window.
F6	Switches the active window.
F7	Marks block start in the editor (Ctrl-K/B).
F8	Marks block end in the editor (Ctrl-K/K).
F9	Performs a MAKE operation (see Section 29.1).
F10	Returns you to the menu bar.
Alt-F1	Displays the last help screen you viewed.
Alt-F3	Brings up the Pick menu (see section 25.4).
Alt-F5	Displays the last output screen in full.
Alt-F9	Compiles the main file.
Alt-F10	Displays the Turbo Pascal title/version box.
Alt-C	Brings up the Compile menu.
Alt-E	Takes you into the editor.
Alt-F	Brings up the Files menu.
Alt-O	Brings up the Options menu.
Alt-R	Runs your program, compiling if necessary.
Alt-X	Quits the Environment and returns you to DOS.
Ctrl-F1	Shows help info for item at editor cursor.

Virtually all dialog boxes have a default response. This is a word or phrase that Turbo Pascal considers the most likely response, and it will be displayed in the dialog box when the box first appears. If the default response is your chosen response, you only need to press Return to complete the dialog. If you wish to enter some other response, simply type it into the dialog box. The default response will vanish as soon as you press the first key that is not the Return, Esc, Backspace, nor an arrow key.

If you find yourself confronted with a dialog box that you don't want to answer, press Esc and you will back out of the dialog.

A *file selection matrix* is a special type of dialog box designed to facilitate selection of one file name from many. It is a large rectangular box within which are displayed rows and columns of file names. One is highlighted with a bounce bar, and you can select one from the matrix by moving the bounce bar with the arrow keys. When the bounce bar highlights the file you want, press Return.

You can navigate through DOS's subdirectories with a file selection matrix. Subdirectory names appear in the matrix as well, and you can tell them from file names because subdirectory names are always terminated with a backslash: \. If you select a subdirectory, the matrix will clear, and Turbo Pascal will read and display the files and subdirectories that exist in the selected subdirectory. You can continue selecting subdirectories and move through the entire DOS directory tree. The full pathname of the displayed subdirectory always appears in the top border line of the selection matrix.

Table 25.2
Turbo Pascal 5.0 Shortcuts

Shortcut Key	*Meaning*
F1	Invokes the Turbo Pascal help system.
F2	Saves the file currently being edited.
F3	Lets you load a new file into the editor.
F4	Execute to cursor position.
F5	Zooms and unzooms the currently active window.
F6	Switches the active window.
F7	Single steps; traces into subprograms.
F8	Single steps; steps over subprograms.
F9	Performs a MAKE operation (see Section 29.1).
F10	Returns you to the menu bar.
Alt-F1	Displays the last help screen you viewed.
Alt-F3	Brings up the **Pick** menu (see section 25.4).
Alt-F5	Displays the last output screen is full.
Alt-F6	Switches between Watch and Output windows.
Alt-F9	Compiles the main file.
Alt-F10	Displays the Turbo Pascal title/version box.
Alt-B	Brings up the **B**reak/Watch menu.
Alt-C	Brings up the **C**ompile menu.
Alt-D	Brings up the **D**ebug menu.
Alt-E	Takes you into the editor.
Alt-F	Brings up the **F**iles menu.
Alt-O	Brings up the **O**ptions menu.
Alt-R	Brings up the **R**un menu.
Alt-X	Quits the Environment and returns you to DOS.
Ctrl-F1	Shows help infor for item at editor cursor.
Ctrl-F2	Performs a **P**rogram reset on program under test.
Ctrl-F3	Displays the **C**all Stack.
Ctrl-F4	Brings up the **E**valuate dialog box.
Ctrl-F7	Adds a watch to the watch window.
Ctrl-F8	Toggles a breakpoint at the cursor line.
Ctrl-F9	Runs your program.

The size of a file selection matrix does not change, and it can at most display 36 file names at one time. If more than 36 files of subdirectories exist in the displayed directory, the message "Too many files" will be displayed in the lower left corner of the matrix border. *This is not an error message.* Nothing has gone wrong. The message only indicates that there are more files than can be shown at once. You can scroll through the rest of the list by using the up and down arrow keys.

As with dialog boxes, you can back out of a file selection matrix by pressing Esc.

25.3: THE FILES MENU

Because files are the only long-term storage a computer has, most computer work centers around creating, selecting, processing, and modifying files. Turbo Pascal is no different, and of its several menus the one you are likely to use most will be the Files menu.

You can bring up the Files menu by selecting it from the menu bar, or by pressing the Alt-F shortcut. In either case, the menu will appear beneath the menu bar, as shown in Figure 25.3. As with all Turbo Pascal menus, you can select an item by moving the bounce bar over the item with the arrow keys and pressing return, or else pressing the first letter of your chosen item.

In the rest of this section, we'll take a close look at each of the Files menu items.

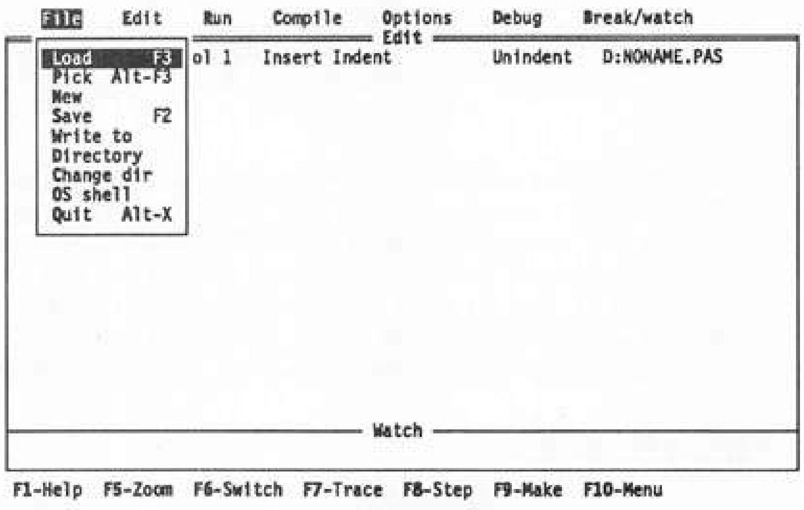
Load

The Load command allows you to choose a file to load from disk into the editor. When you select Load, a dialog box appears atop the Files menu. The top line of the box reads Load File Name, and its job is to accept a file name from you. This name can be of a specific file, or it can use the DOS wildcard characters * and ? to indicate a group of files sharing common name elements.

The best example is the default file name, which is *.PAS. This ambiguous file name matches any file in the current directory with an extension of .PAS. The default file name can be overwritten by any name you choose to enter, or, if you simply press Return, you can use the default.

Figure 25.3

The Files Menu


```
File    Edit    Run    Compile    Options    Debug    Break/watch
                                    Edit
 Load      F3  ol 1   Insert Indent          Unindent   D:NONAME.PAS
 Pick  Alt-F3
 New
 Save      F2
 Write to
 Directory
 Change dir
 OS shell
 Quit   Alt-X

                                   Watch

F1-Help  F5-Zoom  F6-Switch  F7-Trace  F8-Step  F9-Make  F10-Menu
```

After you press Return, one of two things will happen:

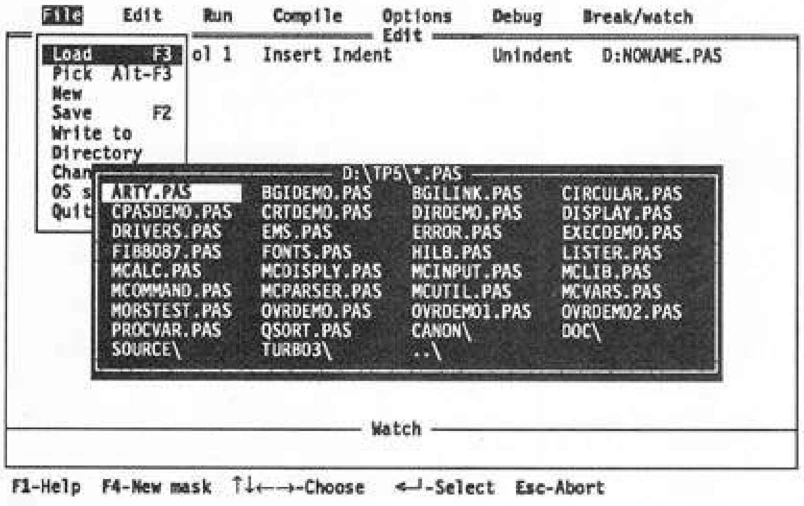
1. If you entered a specific, unambiguous file name, Turbo Pascal will attempt to load it into the editor. If the load is successful, you will be taken into the editor, with the file ready for editing. If the name of the file you entered does *not* exist in the current directory, Turbo Pascal will create the file under that name and put you in the editor to edit the newly-created file.
2. If you entered an ambiguous file name, that is, one incorporating wild card characters, a file selection matrix will appear, displaying all file names that match the ambiguous file name (see Figure 25.4). There is a bounce bar highlighting the file in the upper left corner of the matrix, and you can move the bounce bar around the matrix by using the four arrow keys. When the highlight bar is over the file you want to load, press Return, and the file will be loaded. If you don't want to load any of the files, press Esc.

Pick

One of the cleverer user-interface features I first saw in Turbo Pascal 4.0 was the *pick menu,* which is brought up by selecting **Pick** in the **Files** menu. The pick menu is a short term memory of the last nine files that were edited and saved to disk from the Turbo Pascal editor. This memory is saved to disk when you exit Turbo Pascal, so the next

Figure 25.4

A File-Selection Matrix



```
 File    Edit    Run    Compile    Options    Debug    Break/watch
 ┌─────────────┐══════════════════ Edit ═══════════════════════════
 │ Load     F3 │ol 1   Insert Indent          Unindent   D:NOMANE.PAS
 │ Pick  Alt-F3│
 │ New         │
 │ Save     F2 │
 │ Write to    │
 │ Directory   │
 │ Chan──────────────────── D:\TP5\*.PAS ─────────────────────────
 │ OS s  ARTY.PAS       BGIDEMO.PAS    BGILINK.PAS    CIRCULAR.PAS
 │ Quit  CPASDEMO.PAS   CRTDEMO.PAS    DIRDEMO.PAS    DISPLAY.PAS
 └─────  DRIVERS.PAS    EMS.PAS        ERROR.PAS      EXECDEMO.PAS
         FIB8087.PAS    FONTS.PAS      HILB.PAS       LISTER.PAS
         MCALC.PAS      MCDISPLY.PAS   MCINPUT.PAS    MCLIB.PAS
         MCOMMAND.PAS   MCPARSER.PAS   MCUTIL.PAS     MCVARS.PAS
         MORSTEST.PAS   OVRDEMO.PAS    OVRDEMO1.PAS   OVRDEMO2.PAS
         PROCVAR.PAS    QSORT.PAS      CANON\         DOC\
         SOURCE\        TURBO3\        ..\

 ──────────────────────────── Watch ──────────────────────────────

 F1-Help  F4-New mask  ↑↓←→-Choose  ←┘-Select  Esc-Abort
```

time you turn your computer on and run Turbo Pascal, you can still remember which files you last worked on, even across a hiatus of weeks or months.

Why is this useful? If you are working on a project that incorporates a number of different files, the pick menu will allow you to load each of them quickly, without having to type their names once again. Just bring up the pick menu, bounce the bounce bar to the file name you want to load, and press Return. If the files you are working on are scattered across a subdirectory tree with many levels of nesting, this can save an enormous amount of typing of path names.

If you are simply alternating work on two different files, it's even easier. When the pick menu first appears, the bounce bar will highlight the *last* file you loaded before the current one. If you are only working on two files, this default file will always be the one that *isn't* loaded. So to load your alternate file, simply bring up the pick menu and press Return without moving the bounce bar at all.

The pick menu is saved as a disk file (called the *pick file*) by Turbo Pascal every time you exit the Environment with Alt-X. Apart from the name of the file, the last position of the cursor is saved as well, along with any marked block of text in the file. **P**ick thus becomes a very convenient way to pick up where you left off at the end of a long day. As long as you exit Turbo Pascal through Alt-X (rather than simply rebooting or powering down), your next invocation of TURBO at the DOS prompt will bring you back into the editor at the exact position in the file from which you last exited the Environment.

When the pick file is saved, it is saved into the current directory. This means you can have a separate pick file for every one of your project subdirectories, as long as the TURBO directory containing the compiler itself is somewhere on your DOS path. The secret is to change the current directory to your chosen project directory and *then* invoke Turbo Pascal, rather than invoke Turbo Pascal from the TURBO directory and then change to your project subdirectory.

The name of the pick file defaults to TURBO.PCK. You can change the name of the pick file through the **D**irectories submenu of the **O**ptions menu (see Section 28.3).

The tenth item in the pick menu is always "—load file—" which, when selected, brings up the same dialog box you see when you choose the **L**oad item from the **F**iles menu.

There is a shortcut to the pick menu: Alt-F3.

New

Every so often, you will need to simply begin editing a brand new file that has not yet been created. The **F**iles menu makes this quick and easy with its **N**ew option. **N**ew, when selected, clears the current file from the editor and allows you to edit a brand new file with the default name NONAME.PAS. When you save NONAME.PAS, it will allow you to enter a new name by bringing up a dialog box asking for a new name. If NONAME.PAS is a good enough name for the moment, you can simply use it. Keep in mind that you may have another file on disk named NONAME.PAS, and that saving NONAME.PAS again will erase the old copy.

I recommend changing NONAME.PAS to something more meaningful, to avoid inevitable confusion.

New is careful not to clear out a previous file from the editor if changes have been made to the previous file since the last time it was saved. A dialog box named **Verify** will appear, asking your permission (in the form of a Y/N answer) to the question:

```
OLDFILE.PAS not saved.  Save? (Y/N)
```

Pressing either Y or N will do; you don't have to press Return as well.

There is no shortcut to **New**.

Save

The **Save** item saves the current file to disk under its own name. It is completely equivalent to the shortcut F2. If you wish to write the current file out to a different file name, use the **Write** to item described below.

Write to

You may want to create a second copy of a file, or for whatever reason write the current edit file out to disk under a different name. This is the job performed by the **Write** to Item in the **Files** menu. Write to brings up a dialog box entitled **New Name**. You then respond with whatever name you wish the new file to have. Turbo Pascal then writes out the current file to the new name, *and* makes the new name the current edit file name.

Keep in mind that once you create a second copy of your file with **Write** to, you will continue working with the second copy. Saving to disk saves the file under the new name. To go back to working on the first copy of the file, use the pick menu.

There is no shortcut to **Write** to.

Directory

Choosing the **Directory** item from the **Files** menu brings up a file selection matrix identical to the one brought up by the **Load** item. The difference is that you can't load a file from the matrix, and the default file name is *.*; in other words, all files and all subdirectories (Load defaults to *.PAS).

There is no shortcut to **Directory**.

Change dir

When you invoke Turbo Pascal, its current directory becomes the default directory for file selection through the various menu items. You can change this current directory to some other directory with the Change dir item. A dialog box will appear, allowing you to enter the new directory name. The default name is the name of the current directory, so if you accidentally hit Return before entering a new name, nothing will be changed.

Keep in mind that if you move to a new directory with Change Dir, you carry your current pick file along with you, even if the new directory has its own pick file. Furthermore, if you exit the Environment with Alt-X after having moved to a new directory, you will exit into that new directory, *without* saving a copy of the current pick file into that directory. The pick file is saved back to the same file it originally came from; that is, from the pick file in the directory from which you first invoked Turbo Pascal.

The same is true of options files. You can have a separate options file in every directory, but when you change directories with Change dir you carry your original options with you. To make use of the options in another directory, you must explicitly load them with the **L**oad options item in the **O**ptions menu.

All this may be confusing until you've had a few weeks experience with the Environment. In time it will all seem natural to you. Starting a stick shift car from a dead stop probably seemed confusing at first, but after a few hundred times it gets burned into your synapses and you no longer think about how it works. This will happen with most aspects of using the Environment. Be patient, and use it a *lot*.

OS shell

For all that it does for you, at times the Turbo Pascal Environment will not be enough. For example, you may want to write a source file out to a diskette to give to a friend, but the disk you put in the drive is not formatted. Turbo Pascal can't format disks, but it would be a nuisance to exit the Environment only to have to re-enter it after executing a simple FORMAT command. The **O**S shell item in the Files menu lets you "duck out" to DOS to do a little work, and then pop back into Turbo Pascal just where you left it.

For reasons that would be difficult to explain, many old-timers refer to this process in the generic as "shelling out to DOS" (not the same as "shelling out for DOS," which we do more and more with each new DOS version) Getting back in is as simple as typing the DOS EXIT command.

The **O**S shell item will be most useful if you remember its single great limitation: Turbo Pascal is still in memory and taking up room while you work with DOS. To drive home the point, use **O**S shell to go out to DOS and run the DOS CHKDSK command. As its last display, CHKDSK summarizes how much memory your machine has, total, and then how much of it is available:

```
655350 bytes total memory
158320 bytes free
```

Quite a surprise, isn't it? Given DOS 3.1, Turbo Pascal, Sidekick, and one or two device drivers in memory, this is about what you'll be left with.

The upshot is that you won't be able to run large DOS programs like Paradox or, heaven knows, Ventura Publisher. Also, emphatically *don't* load any memory resident programs! If you do, once you exit Turbo Pascal you'll have split your system memory pretty neatly in half, with a resident utility right in the middle. What DOS will think of that is unclear, but in experiments it's done everything up to and including a total system lockup.

There is no shortcut to **O**S shell.

Quit

The **Q**uit item in the **F**iles menu is completely equivalent to the Alt-X shortcut. Make sure you always exit Turbo Pascal through this menu item or its shortcut, even if all you intend to do after exiting is turn power off. Turbo Pascal does not write the in-memory state of the pick menu to disk until you exit Turbo Pascal. If you simply pull the plug, Turbo Pascal will not be able to pick up your work exactly where you left it.

25.4: A TURBO PASCAL FILES BESTIARY

In its quantum leap from version 3.0 to 4.0, Turbo Pascal traded simplicity for raw power. The power is terrific, once you've learned how to use it, and much of what may be standing in your way is just making sense of the product's new complexity.

Turbo Pascal's treatment of disk files is a particular Pandora's Box of confusion. There are source files, backup files, include files, unit files, pick files, options files, overlay files, .EXE files, assembly language .OBJ files, BGI driver files, BGI font files, and a few more that may or may not be familiar to you. Some have assumed file extensions; most do not.

The only thing they have in common is their potential to confuse the beginner. In this section I'll briefly describe each type of file used by Turbo Pascal 4.0 and 5.0, and try to fit them all into your big picture of the product as a whole.

Source Files

Source files are collections of human-readable statements in the Pascal language. The Turbo Pascal compiler takes these files, analyses them, and based on their contents creates a .TPU file or a .EXE file that implements in machine code the logic you expressed in Pascal.

If you don't enter a file name extension for a source file, Turbo Pascal will append .PAS on the end. To be compilable, a source file must be either a complete Turbo Pascal program or unit.

Include Files

Include files are also source files, but typically they do not comprise a complete program or unit. Usually, include files contain a single function or procedure, or perhaps the definition of a data type. Cutting a subprogram or data definition out into an include file allows more than one program to use the files, with the assurance that each user is getting exactly the same code or data.

Include files are included into a source code file by the use of the **$I** compiler command (see Section 27.3). The default file extension for include files specified without extensions is .PAS. By loose convention, many people use the extension .SRC for include files containing functions or procedures, and .DEF for include files containing definitions of some sort.

Include files were absolutely necessary in the days before Turbo Pascal 4.0, but now, with separately compiled units in our hip pocket, there is very little need for include files anymore. Put such inclusions into units everywhere you can. The reason is simple: Include files must be compiled each time you compile the program as a whole, but a unit is compiled once, and then simply linked into any program that **USES** it. Linking alone is much faster than compiling and linking together.

Backup (.BAK) Files

When you save a source code file out to disk after modifying it, the Turbo Pascal editor does not overwrite the previous version of the file. Instead, it writes an entirely new file out to disk, and renames the previous version by replacing its file extension with the new extension .BAK. Nothing else is changed about the file other than its extension. The *next* time you write the file out to disk, a new copy is created for the current save; the previous version is renamed .BAK, and the original .BAK file is deleted.

The idea here is to always have two, almost identical copies of the same file on disk. That way, if your latest version is written to a portion of disk containing bad sectors and becomes unreadable, you will be able to fall back on the backup copy.

Executable Program (.EXE) Files

When you compile the .PAS source code file of a Turbo Pascal program, what you get is a .EXE file containing executable machine code. These executable program files always have the .EXE extension; Turbo Pascal no longer produces .COM files. An .EXE file can be run right from DOS without any help from Turbo Pascal.

Separately Compiled Unit (.TPU) Files

When compiled, a .PAS source code file arranged as a Turbo Pascal unit will produce a file with an extension of .TPU. This stands for Turbo Pascal Unit. A unit file is not a complete program, but is instead a collection of compiled code, data, or both that may be used quickly and easily by other units or programs without recompiling them.

If you change the file extension of a compiled unit to something other than .TPU, the compiler will not recognize it and you will not be able to use it.

Turbo Pascal Library (.TPL) Files

You will most likely have only one file on your disk with the .TPL extension: TURBO.TPL. This is the system library, always containing the SYSTEM.TPU file, and optionally containing other standard unit files like CRT.TPU, GRAPH.TPU, and PRINTER.TPU. Units residing in TURBO.TPL are loaded into memory when Turbo Pascal is executed, and thus are always available for immediate linking into your code. You can add or remove unit files to TURBO.TPL with the unit mover utility TPUMOVER.EXE.

It is possible to create .TPL files of your own by using TPUMOVER, but as the compiler will not recognize any .TPL file but TURBO.TPL, there's really very little point in it.

Assembly Language External (.OBJ) Files

The **$L** compiler directive (see Sections 24.5 and 27.3) allows you to link assembly language external routines into your Turbo Pascal programs. These routines must be contained in files in the standard Microsoft/Intel .OBJ format, such as that produced by Microsoft's MASM macroassembler. The file extension of such files must be .OBJ or the compiler will reject them.

Pick (.PCK) Files

Each time you exit Turbo Pascal by the **Q**uit item in the **F**iles menu, or through the Alt-X shortcut, Turbo Pascal takes the current pick menu and writes it out as a disk file. The default name of the pick file is TURBO.PCK, but you can change that default to anything you like, including a file with an extension other than .PCK. If you change the name of the pick file, the new name must be saved to disk as part of the options file or the change will not "stick."

There may be a pick file in every directory. I recommend keeping the default name TURBO.PCK (it's easy to recognize) unless you absolutely must change it. We're trying to *lessen* confusion here, no?

Options (.TP) Files

When you reconfigure Turbo Pascal by altering some of the options in the **O**ptions menu, you must save those options out to disk for them to remain in force for the next time you invoke Turbo Pascal. The Save options item in the Options menu saves them to disk. The default file name is TURBO.TP, but you have the option of saving a custom options file under any name you choose. This allows you to have more than one options file (and hence more than one custom setup of Turbo Pascal) in a single directory. You might use one to develop graphics software, with all safety checking disabled, and another to develop business software, with the safety checks in place.

When you invoke Turbo Pascal from a given directory, it will search that directory for a file named TURBO.TP and use it if found. You can also load a custom options file with the Load options item in the Options menu.

Command-Line Compiler Configuration (.CFG) Files

Files with the extension .CFG serve the same purpose to the command-line version of Turbo Pascal as the .TP files serve to the Environment. They contain options specified in human-readable characters as one or more lines of commands beginning with slash characters. If present in a .CFG file, an option overrides the command-line compiler's default value for that option.

The default .CFG is TPC.CFG. If the command line compiler finds TPC.CFG in the current directory, it will open and use it. If not, it will look for TPC.CFG in the directory where TPC.EXE resides. If TPC.CFG is found in neither place, the TPC option defaults will remain in force. You can specify a custom .CFG file to the command line compiler with the /T command on the compile line when invoking TPC.EXE.

Make (.MAK) Files

Turbo Pascal's MAKE utility expects a file containing dependency commands. It looks first for a file called MAKEFILE; not finding that, it will look for MAKEFILE.MAK. If neither is present, it will halt with an error message.

If a file BUILTINS.MAK is present, it will be opened and read before MAKEFILE or MAKEFILE.MAK.

The MAKE utility does not insist on the .MAK extension, but I feel that .MAK makes MAKE files easy to spot and thus lessens confusion.

For brevity's sake I have not covered Turbo Pascal's standalone MAKE utility in this book. See the *Turbo Pascal Owner's Handbook* for more details.

Turbo Pascal Map (.TPM) Files

A .TPM file is a binary (i.e., not human-readable) file containing information about the addresses of entities in the program with the same name but a .PAS extension. The information is used by Turbo Pascal to find the exact location of runtime errors when they occur, and also for symbolic debugging under Turbo Pascal 5.0. The compiler does not ordinarily produce .TPM files. You must turn on .TPM generation by using the Turbo Pascal map file item in the Compiler submenu of the Options menu.

.TPM files may be turned into human-readable .MAP files with the TPMAP.EXE utility.

Symbolic Debugger (.MAP) Files

The .TPM file format mentioned above is specific to Turbo Pascal and is intended to be read by the compiler and nothing else. There is a standard debugger file format used by many symbolic debuggers like Periscope, called .MAP. Unlike .TPM files, .MAP files are human readable and contain some interesting low-level information about a compiled program. You can convert a .TPM file to a .MAP file with Turbo Pascal's TPMAP.EXE utility.

Dependency (.DEP) Files

.DEP files are produced automatically by the TPMAP utility when it converts .TPM files to .MAP files. The .DEP file is a human-readable summary of what uses what and what was linked into what. It is a complete statement of what components are required to rebuild any given program.

Overlay (.OVR) Files (Version 5.0)

Overlays, which were dropped from Turbo Pascal 4.0, were reintroduced with Version 5.0 as an extension to the units architecture for separate compilation. Units are described fully in Section 17, and overlaid units in Section 17.5. Briefly, when overlaid program MYPROG.PAS is compiled, the nonoverlaid portions are compiled into MYPROG.EXE, and the overlaid portions (i.e., all units marked as overlays with **$O**) are compiled to MYPROG.OVR. Keep in mind that a single .OVR file contains *all* overlaid units, and that there may be only one .OVR file per program. The .OVR files contain only code; overlay data is placed in the single Turbo Pascal data segment, just as all data is.

BGI Driver (.BGI) Files

The Borland Graphics Interface supports many different graphics devices by providing a separate set of low-level graphics drawing routines customized for each different graphics device. There is a file for the CGA, for the EGA, for the Hercules card, and so on. These files all have the .BGI extension.

The most common devices are supported by .BGI files included with Turbo Pascal. Some manufacturers of more exotic graphics boards have written their own BGI drivers, and supply a .BGI file on a disk with their graphics boards.

The BGI goes out to the hardware when your program initializes the graphics system, and attempts to detect the installed graphics device. If successful, it loads the appropriate .BGI file into memory (on the heap, in fact) and uses the routines contained in it to display graphics on the detected device. When the graphics system is shut down, the heap memory occupied by the driver is released.

Once you determine which .BGI file actually supports your installed graphics board, you can erase the others from your working directory. This assumes that you have master copies of all Turbo Pascal files somewhere; you're insane if you don't!

BGI Font (.CHR) Files

The BGI supports loadable character fonts of several different types and many styles. These fonts are stored in special files with the .CHR extension. They are loaded at runtime, when specified in a **SetTextStyle** procedure call.

26 Using the Turbo Pascal Editor

Creating a Pascal program involves editing a text file of Pascal statements that the compiler will translate into pure machine code. The Turbo Pascal environment includes a middling powerful screen editor that allows you to create and alter program text files and save them to disk. The editor uses commands that are a subset of those offered by the popular WordStar word processing program. You can, however, redefine the keystrokes for each command to any different set of keystrokes which you prefer or are used to. We will not describe this process here; the TINST program will do this for you. Read up on the process in Appendix F of the *Turbo Pascal Owner's Handbook.*

26.1: THE WORK FILE

All editing is done on what is called the *work file.* Loading a work file is done in one of several ways:

1. *From the command line, when you load the Turbo Pascal Environment.* If you type the name of the file you wish to edit after the program name TURBO, that file will be loaded for edit when the Environment executes. For example,

```
C:\>TURBO SCRIBBLE
```

   will load a file called SCRIBBLE.PAS (if it exists) for edit. If the selected file does not exist, the Environment will create it and allow you to edit the new and empty file.
2. *Through the Load item in the Files menu.* Bringing up the Files menu (Alt-F) and selecting Load will trigger a dialog box asking for the name of a file. You can type the name of a specific file into the dialog box, and that file will be loaded.
   You will notice that the dialog box contains a default file name, which is either the last file name you entered into the dialog box on a previous occasion, or else the ambiguous file name *.PAS. Pressing Return will load the default file name. If the default is an ambiguous file name like *.PAS, a file selection matrix will appear, and you will be able to select one of the files that matches the ambiguous file name by moving the bounce bar over it and pressing Return. Use the arrow keys to move the bounce bar.
3. *Through the Load shortcut F3.* Pressing F3 from anywhere in the environment will bring up the Files menu and select the Load item. The dialog box will appear as described above, ready for your input.
4. *Through the pick menu.* I described the pick menu in some detail in Section 25.3. It is invoked as the Pick item in the Files menu. The pick menu is a menu of the last several files that you edited in the Turbo Pascal editor. You can highlight one of the items with the bounce bar, press Return, and that item will be loaded for editing.

However you choose to select a file the Turbo Pascal environment will attempt to open the file and load it into memory. If the file already exists on disk the entire file will be loaded into memory. If the file does *not* exist, the Turbo Editor will create the file under the name you entered. If you misspell the name of the file you want to load, the editor will gladly go out and create a brand new empty file for you with the misspelled name. Watch your typing!

Editor work files are limited to 64K in size, and must be able to be loaded into memory in their entirety. If the file is too large to fit in available memory, you will see this message:

```
File too big. Press <ESC>
```

Pressing Esc will back out of the file load operation and allow you to enter another file name.

Also keep in mind that individual lines within an edit file are limited to 248 characters. Loading a file with longer lines will cause the editor to insert hyphens at the 248-character point.

26.2: MOVING THE CURSOR

Apart from Environment shortcut keys that are still valid within the editor, all editor commands are control keystrokes; that is, you must hold the control (Ctrl) key down while pressing another key or two keys. All of the keys which control cursor movement are grouped together for you in a cluster toward the left hand side of the keyboard:

```
  W   E   R
A   S   D   F
  Z   X   C
```

This arrangement of cursor command keys will be familiar to anyone who has worked with the WordStar word processor.

One Character at a Time

Moving the cursor one character at a time can be done in all four directions:

CTRL-E or **Up Arrow** moves the cursor **Up** one character.
CTRL-X or **Down Arrow** moves the cursor **Down** one character.
CTRL-S or **Left Arrow** moves the cursor **Left** one character.
CTRL-D or **Right Arrow** moves the cursor **Right** one character.

The position of these four keys (E, X, S, and D) provide a hint as to which way they move the cursor. Look at how they are arranged on the keyboard:

```
   E
S     D
   X
```

Until the directions become automatic to your fingers (as they will, if you do enough editing!) thinking of the "magic diamond" will remind you which way the cursor will move for which keypress.

When you move the cursor to the bottom of the screen and press Ctrl-X one more time, the screen will *scroll.* All the lines on the screen will jump up by one, and the top line will disappear. As long as the cursor is on the bottom line of the screen and you continue to press Ctrl-X, the screen will scroll upward. If you use Ctrl-E to move the cursor back in the opposite direction (upward) until it hits the top of the screen, further Ctrl-E's will scroll the screen downward one line per Ctrl-E.

One Word at a Time

The Turbo Editor will also move the cursor one word at a time to the left or right:

CTRL-A or **CTRL-Left Arrow** moves the cursor **Left** one word.
CTRL-F or **CTRL-Right Arrow** moves the cursor **Right** one word.

More hints are given here, since the A key is on the left side of the magic diamond, and the F key is on the right side of the magic diamond.

One Screen at a Time

It is also possible to move the cursor upward or downward through the file one whole screen at a time. *Upward* in this sense means toward the beginning of the file; *downward* means toward the end of the file. A screen is the height of your CRT display (25, 43, or 50 lines, depending on Environment options) minus three lines for the editor status line and the Environment's menu and prompt bars.

CTRL-R or **PgUp** moves the cursor **Up** one screen.
CTRL-C or **PgDn** moves the cursor **Down** one screen.

Moving the Cursor by Scrolling the Screen

We described how the screen will scroll when you use the one-character-at-a-time commands to move upward (Ctrl-E) from the top line of the screen or downward (Ctrl-X) from the bottom line of the screen. You can scroll the screen upward or downward no matter where the cursor happens to be by using the scrolling commands:

CTRL-W scrolls the screen **Down** one line.
CTRL-Z scrolls the screen **Up** one line.

When you scroll the screen with these commands, the cursor "rides" with the screen as it scrolls upward or downward, *until* the cursor hits the top or bottom of the screen. Then further scrolling will make the screen "slip past" the cursor. The cursor will always remain visible.

These are all of the cursor movement commands that may be invoked by one control keystroke. There are a few more that are accomplished by holding the Control key down and pressing *two* keys in succession. *You must hold the Control key down through both keypresses!*

Moving to the Ends of the Line

No matter where your cursor is on the screen, it is always within a line, even if that line happens to be empty of characters. There are two commands that will move the cursor either to the beginning (left end) of the line (screen column 1) or to the end of the line, which is the position following the last visible character on the line:

CTRL-Q/S or **Home** sends the cursor to the **Beginning** of the line.
CTRL-Q/D or **End** sends the cursor to the **End** of the line.

Moving to the Ends of the File

The last set of cursor movement commands we'll describe takes the cursor to the beginning of the file or to the end of the file. If the file you are editing is more than a few screens long, it can save you a great deal of pounding on the keyboard to move one screen at a time.

CTRL-Q/R or **CTRL-PgUp** sends the cursor to the **Beginning** of the file.
CTRL-Q/C or **CTRL-PgDn** sends the cursor to the **End** of the file.

Because all of your file is in memory all of the time, moving between the ends of the file can be done *very* quickly.

Moving to the Last Error Position

A special command within the editor allows you to move the cursor to the last identified error position:

CTRL-Q/W sends the cursor to the **Last Error** position.

26.3: THE EDITOR STATUS LINE

If you did any practicing at all in moving the cursor around a test file, you may have noticed the line at the top of the screen, just below the menu bar. This line did not move no matter where you sent the cursor. This is the editor status line, and it provides you with some important information while you are editing.

A typical instance of the status line for version 4.0 looks like this:

```
Line 1    Col 1    Insert Indent      C:FOO.PAS
```

The Turbo Pascal 5.0 editor status line is a little different:

```
Line 1    Col 1    Insert Indent       Unindent * C:FOO.PAS
```

While you were moving the cursor around, the line and column numbers were continually changing to reflect where the cursor was in the file. The column number reflects the position of the cursor within its line; the line number indicates which line in the file contains the cursor, counting from the beginning of the file, *not* from the top of the screen.

At the other end of the status line is the name of the current work file, which is the name of the file you are editing.

The other words (Insert, Indent, and Unindent) shown as part of the status line merit some explaining. Insert and Indent are the names of two "toggles". A toggle is a condition which may exist in one of two different states. A toggle is like a switch controlling the lights in a room; the switch may be either on or off.

Insert determines how newly typed characters are added to your work file. When Insert is on (that is, when the word Insert appears in the status line) characters which you type are *inserted* into the file. The characters appear over the cursor and immediately push the cursor and the rest of the line to the left to make room for themselves. The line becomes one character longer for each character that you type. If you press Return, the cursor moves down one line, carrying with it the part of the line lying to its right.

When Insert is off (if the word Insert is *not* displayed in the status line) characters that you type will *overwrite* characters that already exist in the file. No new characters are added to the file unless you move the cursor to the end of the line or the end of the file and keep typing. If you press Return, the cursor will move down to the first character of the next line down, but nothing else will change. A line will only be added to the file if you press Return with the cursor on the last line of the file.

Turning Insert on and off is done with a single control keypress:

CTRL-V toggles **Insert** on and off.

Indent is also a toggle. It indicates whether the Turbo Editor's auto-indent feature is on or off. When indent is on, the cursor will automatically move beneath the first visible character on a new line when you press Return. In other words, (assuming that Indent is on) given this little bit of text on your screen:

```
FOR I := 1 TO 10 DO
  BEGIN
    Total := Total + I;
    Count := Succ(Count);_   <--Before pressing Return
    _
    ^
    | After pressing Return
```

The cursor is at the end of the last line of text. When you press Return, the cursor will move down one line, but it will also space over automatically until it is beneath the C in **Count**. This allows you to begin typing the next line of code without having to space the cursor over so that it is beneath the start of the previous line.

The editor's Indent feature allows easy "prettyprinting" of your Pascal source code files. This makes for easier reading of your source code files, and easier entry when you go to type them in.

Like Insert, Indent can be toggled on and off. It takes a double control keystroke to do it:

CTRL-Q/I toggles **Indent** on and off.

Indent is considered on when the word Indent appears in the status line.

Unindent was added with Turbo Pascal 5.0, and will not be seen in the status line for version 4.0. It is quite literally the opposite of the editor's autoindent feature. When Unindent is on, pressing the backspace key will move the cursor *back* (leftward) to line up beneath the next level of indentation *if* the cursor is on a blank line.

This is easier to show than to describe. Let's go back to the example used to demonstrate autoindent:

```
FOR I := 1 TO 10 DO
  BEGIN
    Total := Total + I;
    Count := Succ(Count);
_ _ _ <--Before pressing Backspace
^ ^
| |-----After pressing backspace once
|-------After pressing backspace twice
```

In this example, the cursor is lined up beneath the C in **Count**, having autoindented in after the user pressed Return with the cursor at the end of the previous line. Press backspace *once*, and the cursor moves back *two* spaces, to line up beneath the **BEGIN** keyword that represents the previous level of indentation. Press backspace a second time, and the cursor will move two more spaces to the left, to line up beneath the **FOR** reserved word.

The uses of Unindent should be obvious: It makes it easy to cap off a **BEGIN..END** block with a nicely indented **END** that lines up under its partner **BEGIN**.

Unindent is toggled on an off, like Insert and Indent, with a double control command. Note, however, that it is a Ctrl-O and not a Ctrl-Q lead-in!

CTRL-O/U toggles **Unindent** on and off.

The "File Modified" Indicator (Version 5.0)

Turbo Pascal 5.0 displays an asterisk on the status line when any modification at all has been made to the edit file since it was last saved. Even if you only type a space, or delete a character and immediately type the same character in its place, the asterisk symbol will appear on the status line, indicating that you had better save the edit file before doing anything adventurous like shelling out to DOS.

Tab Mode

The editor status line also displays the current tab mode. This is a feature new to Turbo Pascal 4.0, and did not exist in earlier versions of the editor.

Beginning with Turbo Pascal 4.0, there are two kinds of tabs in the editor. The 3.0-style tabs are not tabs as most people knew them prior to the onset of Turbo Pascal. These "smart" tabs move the cursor to the position beneath the start of the next word on the previous line. In other words, if you had the following line on your screen with the cursor on the line beneath it, the caret marks show where the cursor would pause at each successive press of the Tab key:

```
Think of it as evolution in action...
      ^  ^  ^  ^         ^  ^
```

This tabbing is done by inserting spaces, *not* by inserting the ASCII Tab (Ctrl-I) character.

Smart tabs as described above are the default tab mode in the editor. Pressing Ctrl-O/T toggles to the opposite tab mode, which supports true, 8-character fixed tabs that insert Ctrl-I characters at each press of the Tab key. If fixed tabs are in effect, the word Tab will be shown on the status line between the word Indent and the filename:

```
Line 1    Col 1   Insert Indent Tab C:MY_PROG.PAS
```

In summary on tab mode,

CTRL-O/T toggles between **Smart Tabs** and **Fixed Tabs**.

26.4: INSERTS AND DELETES

We've already seen how to insert characters into a text file: You make sure Insert is on, and type away. Each typed character will be inserted into the file at the cursor position.

It is also possible to insert entire blank lines. One way, of course, is to move the cursor to the beginning of a line and press Return. (Insert must be on.) A new blank line will be inserted above the line with the cursor, and the rest of the file will be pushed downward. The cursor will ride down with the text pushed downward.

The other way to insert a line is independent of the Insert toggle. Move the cursor to the beginning of a line and press Ctrl-N. A new line will appear, pushing the rest of the file downward, *but the cursor will not move down with the other text.*

CTRL-N inserts a **New line** at the cursor position.

There are also a number of different ways to *delete* text as well. The simplest is to use the Delete key:

Delete deletes one **Character** to the **Left** of the cursor.

The cursor moves one character position to the left with each press of the Delete key. Notice that the Backspace key does not delete anything; it merely moves the cursor leftward, just as Ctrl-S moves the cursor leftward.

Deleting characters to the right of the cursor is done a little differently:

CTRL-G deletes one **Character** to the Right of the cursor.

The cursor does not move. It "swallows" the character to its right, and the rest of the line to its right moves over to fill in the position left by the deleted character.

You can also delete (to save a few keystrokes) one word to the right of the cursor:

CTRL-T deletes one **Word** to the **Right** of the cursor.

When you press Ctrl-T, all characters from the cursor position rightward to the end of the current word will be deleted. If the cursor happens to be on a space or group of spaces between words, that space or spaces will be deleted up to the beginning of the next word.

It is possible to delete from the cursor position to the end of the cursor line:

CTRL-Q/Y deletes from the cursor to the end of the line.

And finally, it is possible to delete the entire cursor line with a single control keystroke:

CTRL-Y deletes the entire **Line** containing the cursor.

The line beneath the cursor moves up to take the place of the deleted line, pulling up the rest of the file behind it.

Undoing Changes to a Line

The editor keeps a "backup" copy of each line while your're working on it, and retains that copy as long as the cursor remains within the line. Therefore, if you delete a word or some other portion of the line, or add something to a line by mistake, you can undo those changes to the line *as long as you haven't yet left the line.* Once you leave the line even momentarily, the editor throws away the backup copy, and Undo is no longer possible.

CTRL-Q/L restores a line to its condition before you entered it.

One drawback is that the undo feature will not restore a line deleted entirely with the Ctrl-Y command. Once a line is deleted, the cursor, of necessity, leaves the line, and so the editor does not retain the backup copy of the line. Be careful how you use Ctrl-Y!

26.5: MARKERS AND BLOCKS

The editor supports two different kinds of markers; that is, positions in the file that have a name or number and may be moved around as needed by the programmer. These are *place markers* and *block markers.*

Place Markers

There is no such thing as a page number in an editor file. You can move the cursor to the beginning or end of the file with a single command, but to move to a specific place in the file is harder. The best way is to remember a distinctive title, procedure name, or something like that and search for it (see below). You might also make use of the editor's place marker feature.

The editor supports four place markers, numbered 0 to 3. These may be placed at any position in a text file with a single command:

CTRL-K<n> sets marker <n>within a file. **<n>** may be **0-3**.

For example, to set marker 2, you would press Ctrl-K2.

Place markers can save you a lot of searching and paging through long source files. Perhaps you are working on a source file and decide to use a particular variable. You don't remember, however, whether you declared the variable up in the variable declaration part of the program. Rather than just use the variable and perhaps forget to declare it later, you want to Ctrl-Q/R to the beginning of the file, declare the variable, and then jump right back and pick up what you were doing. The fastest way is to set a marker to your current position, move to the beginning of the file, declare your variable, and then move the cursor back to the marker that you placed back down where the real work is going on.

It's easy. Once a place marker has been set, you can move the cursor to it with a single command:

CTRL-Q<n> moves the cursor to marker **<n>**.

For example, to move to marker 2 you would type Ctrl-Q2. If you have two or three "construction zones" within a largish source file, you might drop one of the place markers at the start of each zone, so you can shuttle between the zones with a single command.

The markers are invisible, and if you forget where they are about all you can do is move the cursor to them with the Ctrl-Q<n> command.

Block Markers

The other kind of marker is used to specify the beginning and end of a text block. There are only two of these markers, and in consequence only one block may be marked within a file at any given time. These are called the block markers, and their names are B and K, after the commands that position them in your file.

The block markers are invisible and do not appear on your screen in any way. If both are present in a file, however, all the text between them (the currently marked block) is shown as highlighted text.

Placing each block marker is a two-character control keystroke:

CTRL-K/B places the **B** marker. The *version 4.0* shortcut is **F7**.
CTRL-K/K places the **K** marker. The *version 4.0* shortcut is **F8**.

Note the two shortcuts for Turbo Pascal 4.0, which are extremely convenient and fast. Sadly, the F7 and F8 shortcuts were pre-empted for the single-step commands in Turbo Pascal 5.0, so in that version blocks need to be marked using the full Ctrl-K/B and Ctrl-K/K commands.

A marker is placed at the cursor position and remains there until you move it elsewhere. You cannot delete or remove a marker once placed, although you can *hide* the block of text which lies between the markers.

Moving the Cursor to a Block Marker

There are also commands to move the cursor to the block markers, B and K:

CTRL-Q/B moves the cursor to the **B** marker.
CTRL-Q/K moves the cursor to the **K** marker.

Hiding and Unhiding Blocks of Text

The major use of markers, however, is their ability to define a block a text. There are a number of commands available in the Turbo Editor which manipulate the text that lies between the B and K markers.

You probably noticed while experimenting with setting markers that as soon as you positioned *both* the B and K markers in a file, the text between them became highlighted. The highlighted text is a marked text block. As we mentioned before, there is no way to remove a marker completely from a file once it has been set. You can, however, suppress the highlighting of text between the two markers. This is called "hiding" a block:

CTRL-K/H will **Hide** a **Block** of text.

Remember that the markers are still there. Ctrl-K/H is a toggle. You invoke it once to hide a block, and you can invoke it a second time to "un-hide" the block and bring out the highlighting again on the text between the two blocks.

Something else to keep in mind: The other block commands we'll be looking at below work *only* on highlighted blocks. Once a block is hidden, it is hidden from the block commands as well as from your eyes.

Marking a Word as a Block

The editor includes a short form of the command sequence for marking a single word as a block. Ordinarily, you'd have to move the cursor to the beginning of the word, press F7, (or CTRL-K/B), then move to the end of the word and press F8 (or CTRL-K/K). With one command you can mark a word as a block, once you've moved the cursor to any position within that word:

CTRL-K/T marks the **word** containing the cursor as a **block**.

Sadly, parentheses, commas, and other punctuation marks are considered word boundaries, so you can't mark an entire procedure header with Ctrl-K/T. In other words, given this line:

```
GotoXY(61,14);
```

If you place the cursor on the G and type Ctrl-K/T, only the word "GotoXY" will be marked.

Block Commands

The simplest block command to understand is delete block. Getting rid of big chunks of text which are no longer needed is easy: Mark the text as a block with the two markers, and issue the delete block command:

CTRL-K/Y will **Delete** a **Block** of text.

The markers do not vanish with the block of text. They "close up" and occupy the same cursor position, but they are still there, and you can move the cursor to them with the Ctrl-Q/B or Ctrl-Q/K commands.

Copy block is useful when you have some standard text construction (a standard boilerplate comment header for procedures, perhaps) which you need to use several times within the same text file. Rather than type it in each time, you type it once, mark it as a block, and then copy it from the original into each position where you need it. Simply put the cursor where the first character of the copied text must go, and issue copy block:

CTRL-K/C will **Copy** a **Block** of text to the cursor position.

Moving a block of text is similar to copying a block of text. The difference, of course, is that the original block of text which you marked vanishes from its original position and reappears at the cursor position. You must first mark the text you wish to move as a block. Then put the cursor where you wish the marked text to go, and issue move block:

CTRL-K/V will **Move** a **Block** of text to the cursor position.

The last two block commands allow you to write a block of text to disk, or to read a text file from disk into your work file. Writing a block to disk begins by marking the block you want saved as a separate text file. Then issue the write block command:

CTRL-K/W will **Write** a **Block** of text to disk.

The editor needs to know the name of the disk file into which you want to write the marked block of text. It prompts you for the file name with a dialog box entitled "Write Block To File." You must type the name of the file, with full path if you intend the block to be written outside of the current directory. Once the name is typed and you have pressed Return, the block is written to disk. The block remains highlighted, and the cursor does not move.

Reading a text file from disk into your work file is also easy. You position the cursor to where the first character of the text from the file should go, and issue read block:

CTRL-K/R will **Read** a **Block** of text from disk to the cursor position.

Just as with write block, the editor will prompt you for the name of the file that you wish to read from disk with a dialog box entitled "Read block from file."

There is one small "gotcha" that you must be aware of in connection with file names. If you enter a file name *without* a period or file extension (that is, a file name like FOO rather than FOO.PAS) the Turbo Editor will first look for a file named FOO. If it does not find one, it will then look for a file named FOO.PAS. If it cannot find a file with the entered name plus the .PAS file extension, it will issue this error message within an alarming red (if you have a color screen) box:

Unable to open FOO.PAS. Press <ESC>

Pressing Esc cancels the command entirely, and to enter the name correctly you will need to issue the Ctrl-K/R command again.

The text file will be read and inserted into your work file at the cursor position. It will come in as a marked block, and you will have to issue the hide block command to remove the highlighting. Remember also that reading a block of text from disk will effectively move your two block markers from elsewhere in your file and place them around the text which was read in.

The editor is not especially picky about the type of files you read from disk. Text files need not have been generated by Turbo Pascal's editor. Files need not be text files at all, in fact; but remember that reading raw binary data into a text file can cause the file to appear foreshortened: The first binary 26 (Ctrl-Z) encountered in a text file is assumed to signal the end of the file. Data after that first Ctrl-Z may or may not be accessible. Furthermore, the editor will attempt to display the binary characters as-is, and loading (for example) an .EXE file will fill the screen with some pretty lively garbage.

26.6: SEARCHING AND REPLACING

Much of the power of electronic text editing lies in its ability to search for a particular character pattern in a text file. Furthermore, once found, it is a logical extension of the search concept to replace the found text string with a different text string. For example, if you decide to change the name of a variable to something else to avoid conflict with another identifier in a program, you might wish to have the text editor locate every instance of the old variable name in a program and replace each one with the new variable name.

The editor can do both search and search/replace operations with great ease. Simply locating a given text string in a program is often better than having page numbering (which the editor does not). If you wish to work on the part of a program that contains a particular procedure, all you need do is search for that procedure's name and you will move the cursor right to the spot you want:

CTRL-Q/F will **Find** a given text string.

When you issue the find command, the editor prompts you with a single word:

```
Find:
```

You must then type the text string that you wish found, ending it with Return. The editor then prompts you for command options:

```
Options:
```

There are several command options which may be given to both the find and find/replace commands. These are single letters (or numbers), and may be grouped together in any order without spaces in between:

```
Options: BWU
```

for example. We'll be discussing each option in detail shortly. Once you press Return after entering the options (if any) the editor executes the command. For the find command, the cursor will move to the first character of the found text string. If the Turbo Editor cannot find any instance of the requested text string in the work file, it displays this message:

```
Search string not found. Press <ESC>
```

You must then press Esc to continue editing.

Find/Replace

Find/replace goes that extra step further. Once the search text is found, it will replace the search text with a replacement text. The options mean everything here; you can replace only the first instance of the search text, or all instances, and you can have the editor ask permission before replacing, or simply go ahead and do the deed to as many instances of the search text as it finds. This last operation is especially beloved of programmers, who call it a "search and destroy."

As with find, the editor prompts for the search text and options. It must also (for find/replace) prompt for the replacement string:

```
Replace with:
```

If no options are in force, the editor will locate the first instance of the search string, place the cursor beneath it, and give you the permission prompt:

```
Replace (Y/N):
```

If you type a Y here (no Return required), the editor will perform the replacement. If you type an N, nothing will change.

Search/Replace Options

The editor's find/replace options allow you to "fine-tune" a find or find/replace command to cater to specific needs. For example, without any options the find command is case-sensitive. In other words, FOO, foo, and Foo are three distinct text strings, and searching for FOO will not discover instances of foo. With the U option in force, however, FOO, foo, and Foo are considered identical and searching for any of the three forms will turn up instances of any of the three which are present. There are several such options to choose from within the editor. In general they are the same find/replace options used by WordStar.

B is the **Search Backwards** option. Ordinarily, a search will proceed from the cursor position toward the end of the file. If the object of the search is closer to the beginning of the file than the cursor, the search will not find it. With the B option in force, the search proceeds *backwards* through the file, toward the beginning.

G is the **Global Search** option. As mentioned above, searches normally begin at the cursor position and proceed toward one end of the file or the other, depending on whether or not the B option is in force. With the G option in force, searches begin at the beginning of the file and proceed to the end, irrespective of the cursor position. The G option overrides the B option.

N is the **Replace Without Asking** option. Without this option, the editor (during a find/replace) will prompt you for a yes/no response each time it locates an instance of the search text. With N in force, it simply does the replacement. Combining the G and N options means that the editor will search the entire file and replace every instance of the search text with the replacement text, without asking. *Make sure you set it up right,* or you can cause wholesale damage to your work file. In general, *don't use G and N together without W* (see below for details on the W option).

U is the **Ignore Case** option. Without this option, searches are case-sensitive. FOO and foo are considered distinct and searching for one will not find the other. With the U option in force, corresponding lower and upper case characters are considered identical. FOO and foo will both be found on a search for either.

W is the **Whole Words** option. Without this option, the search text will be found even when it is embedded in a larger word. For example, searching for LOCK will find both BLOCK and CLOCK. With W in force, the search text must be bounded by spaces to be found. This option is especially important for global search/replace commands, when (if you omit W) replacing all instances of LOCK with SECURE will change all instances of BLOCK to BSECURE and all instances of CLOCK to CSECURE.

You may also give a number as one of the options. For the find command, this tells the editor to find the nth instance of the search text. For find/replace, a number tells the editor to find and replace text n times.

Find or Find/Replace Again

The editor remembers what the last find or find/replace command was, search text, replacement text, options, and all. You can execute that last find or find/replace command again simply by issuing the find or find/replace again command:

CTRL-L will perform the last find or find/replace command **Again**.

Ctrl-L can save you some considerable keystroking. Suppose, for example, you wanted to examine the header line of every procedure in a large (perhaps 1000 line) program with thirty or forty procedures. The way to do it is to search for the string PROCEDURE with the G, U, and W options in effect. The first invokation of this command will find the first procedure in your program file. To find the next one, simply press Ctrl-L. You need not re-enter the search text PROCEDURE or the options. Each time you press Ctrl-L, the editor will find the next instance of the reserved word PROCEDURE until it runs out of file, or until you issue a new and different find or find/replace command.

26.7: SAVING YOUR WORK

It is *very* important to keep in mind what is happening while you edit text files with the editor. *You are editing entirely within memory.* Nothing goes out to disk while you are actually doing the edit. You can work on a file for hours, and one power failure will throw it all away. You must develop the discipline of saving your work every so often.

You can, of course, exit the editor, bring up the **F**iles menu, and select the **S**ave item. That's far more trouble than necessary, and it requires you to exit the editor. The easiest way to execute a Save command from within the editor is with the Save shortcut, F2. F2 will save the editor work file no matter what part of the Environment you happen to be in when you press it.

The "longcut" to saving the file from within the editor is Ctrl-K/S, but F2 is easier to type and remember.

F2 *always* **Saves** your work file.

You can configure the Environment to save you work automatically whenever you execute the **R**un command or duck out to DOS through the **O**S shell item in the **F**iles menu. This is a good idea, since programs under construction sometimes lock up the system, which would prevent you from saving the source file. Also, once you duck out to DOS, Turbo Pascal isn't responsibile for keeping the system alive, so saving the source before shelling out is a wise thing to do.

Enabling this automatic save is done by toggling the **E**dit auto save item on the **E**nvironment submenu within the **O**ptions menu.

Exiting the Editor

There is more than one way to get out of the editor once you're finished with the job at hand. You can move directly from the editor to any of the menus on the main menu bar by using the Alt-key shortcuts. In other words, you can move directly from the editor to the **O**ptions menu by pressing Alt-O, or to the **F**iles menu by pressing Alt-F.

You do *not* have to explicitly exit the editor before compiling or running your program. Pressing Alt-R or Alt-C from within the editor will take you through the compile process directly.

Exiting the Turbo Editor can also done with either of these commands:

CTRL-K/D or **CTRL-K/Q** end the edit and **Exit** to the menu bar.

You can also leave the Environment entirely from within the editor, using the Alt-X shortcut for **Q**uit. If you have modified the work file since the last time you saved it, the editor will ask you if you wish to save the file before exiting.

27 Using the Turbo Pascal Compiler

In a sense, everything in Turbo Pascal exists to support the compiler. The compiler is very much the heart of the product and the process of creating native code programs in Pascal. In this section we'll look closely at everything you need to know about the compiler, its commands, and the options it recognizes.

27.1: WHAT THE COMPILER DOES

The central purpose of the entire Turbo Pascal Environment, of course, is to create Pascal programs. Turbo Pascal compiles your work file to pure native code that runs without the assistance of an intermediate code interpreter in the manner of UCSD Pascal or BASICA. For most program development you can compile your work file (which already exists in RAM) directly to native code in RAM, a process which happens with such speed as to astonish any programmer who has previously worked only with disk-to-disk filter-type compilers.

Once your program has compiled to RAM it can be run from the Turbo Pascal Environment, which "steps aside" while your program executes, and when the program terminates, the Environment reappears. It is thus possible to develop programs without any disk access at all, which is both dizzyingly fast and somewhat dangerous. A frantic session of edit/compile/run, edit/compile/run, can load your RAM memory with irreplaceable code existing nowhere else; perhaps not on paper, in your notes, or (at 3:00 A.M. after 10 hours at the tube) even in your head. Working with Turbo Pascal is undeniably intoxicating, and the tendency to work without ever saving to disk is strong.

Avoid that temptation.

One power glitch, one blown fuse, one little brother tripping over your machine's power cord and yanking it out of the wall will send all your hard work into the mystical bit bucket in the sky. Be wise. Save your program source work file to disk after each modification. It's as easy as pressing F2. And back up your disk after every session.

27.2: THE COMPILE MENU

No matter where you are in the Environment, pressing Alt-C will bring up the Compile menu. The top item in the compile menu is Compile, and by selecting **Compile** your program will be compiled and made ready to run. But before we dive into the Compile menu, there is a shortcut.

The Run Command

The Version 4.0 **Run** keyword has no menu behind it. If you select **Run**, either by pressing Return when it is highlighted, or pressing Alt-R from some other part of the Environment, two things will happen without further assistance from you:

1. Your program will be compiled. A *compile status box* will appear in the middle of your screen, providing a running tally of lines compiled. It goes fast. On a 386 machine like mine, it often happens too quickly to follow.
2. The compiled program will run. The screen will clear, and your program will be given full control. Once it runs its course, the following message will appear at the bottom of the screen:

```
Press any key to return to Turbo Pascal
```

Press any key, and the Environment is as good as its word; Turbo Pascal will reappear, ready for your next program modification. Version 5.0 does not pause for a keypress but returns control immediately to the environment.

Note: If you're running Turbo Pascal 5.0, there *will* be a menu behind the **R**un keyword. The items on this menu, however, all involve debugging, which I'll cover in detail in Chapter 30 and won't go into here.

The Compile Item

Assuming that the Pascal source code in your work file has no errors, compilation by the Turbo Pascal Compiler is a remarkably simple thing. When selected, the Compile item begins compilation on the work file.

While compilation is underway, the compiler displays a compile status box in the middle of the screen. The running tally of lines compiled is there to give you some flavor for how far along it is (and, not coincidentally, how fast it is going.) If the compilation completes without any errors, the compiler will leave the compile status box on the screen, with its summary of the lines compiled and the amount of available memory. To continue, press any key.

Whoops!

No matter how good a programmer you will become, you will write programs with errors. Turbo Pascal is quite helpful in spotting errors in your programs. When the compiler discovers an error in your source code, it will stop the compilation and print a message similar to this in a highlighted line (in red if you have a color screen):

```
Error 85: ";" expected.
```

Beneath the error message line, the editor will be displaying your source code file with the cursor at the location where it noticed the error. *This is not necessarily where the problem exists,* but rather is always where the compiler first suspects that something is wrong.

For example, if you leave out a **BEGIN** in the middle of the program, the compiler will not necessarily notice that a **BEGIN** is missing until further down in the file, when something else doesn't add up. It may be that it finds, at the bottom of the program, that it has one too many **END**s. There are a fair number of other ways in which a missing **BEGIN** will make itself known to the compiler; which one eventually tips the compiler off will depend on how your program is laid out.

The point of all this is that while the compiler will give you a good hint when it passes the baton to the editor and places the cursor where it noticed a problem with your program, you may in fact have to do some further hunting to find the real cause of the problem.

Running Your Compiled Programs from Memory

If you have enough room to compile your programs directly into memory, the Environment will allow you to run your program with just one keypress. The Alt-R command initiates the running of your code. (Note that **R**un is not an item in the **C**ompile menu, but a separate key word on the main menu bar.) At that point, Turbo Pascal saves the current state of the Environment and its screen into a special buffer, brings back the DOS screen that was displayed when Turbo Pascal was originally invoked, and then hands control over to your compiled code.

When your program stops running (when the code executes a **Halt** statement or when execution runs off the bottom end of the program) you'll see this prompt (under Version 4.0 only) at the bottom of the screen:

```
Press any key to return to Turbo Pascal
```

One keystroke of any flavor will bring back the Environment and the screen just as it was when your program took over. Of course, if your program does something drastic or sneaky to system resources, the Turbo Environment may not be able to take control back, and you may have to reboot. This is another good reason to hit F2 *every time* before running one of your programs, or else enable the Edit auto save item on the **O**ptions menu (see Section 28.2) in order that the Environment will automatically save your work file before running your compiled code.

Specifying Command-Line Parameters for In-Memory Testing

Command-line parameters are data you type after the name of a program on the DOS command-line. In other words, in this invocation of a program:

```
C:\>MERGE FILE1.NAP FILE2.NAP
```

the command-line parameters are FILE1.NAP and FILE2.NAP. Turbo Pascal programs can access the command line parameters through the predefined variables **ParamStr** and **ParamCount** (see Section 15.3).

But in testing your Turbo Pascal programs within the Environment, you need a way to provide the program with command-line parameters even though you're not invoking the program under test from the command line.

Turbo Pascal lets you enter command-line parameters through the **P**arameters item in the **O**ptions menu. When you select **P**arameters, a dialog box will appear, labeled Command-Line Parameters. You may enter up to 128 characters, exactly as you would on the command line, with the important exception that you should not also type the name of the program you're testing. In other words, if we keep with the example above, what you would enter into the dialog box is:

```
FILE1.NAP FILE2.NAP
```

Once entered, the parameters will be passed to your program each time you run it, and you needn't type them again. In fact, if you save your options for each important programming project you work on, and load the proper options file before starting work, you can load the command-line parameters along with all the other options. More information about loading and saving options is given in Section 28.4.

The Compile Destination

Immediately beneath the **C**ompile item are the **M**ake and **B**uild items, which I won't discuss now. They are part of Turbo Pascal's project management feature, and I'll take them up in Section 29.1.

Next on the menu is **D**estination, which is a toggle that directs the output of the compiler to either memory or a disk file. The default state is **M**emory. This makes for very fast compiles, and when your program is fairly small, there's very little need initially to create a separate disk file for the executable code.

Problems will appear when your program starts to get ambitious. The Turbo Pascal Environment takes a fair amount of space in memory, since the executable file is 120K-130K depending on the release, with an additional 64K of memory allocated to the editor work file alone. (Version 5.0 will place the editor buffer in EMS memory if the Environment detects it in your system.) Once your program starts to push thirty or forty thousand lines of code (and this may happen sooner than you think, if you use a lot of units) you may have to abandon in-memory compiles.

Changing the **D**estination item to **D**isk writes an .EXE file to the current directory *unless* a directory has been specified for executable files in the **D**irectories submenu of the **O**ptions menu (see Section 28.3).

Finding Runtime Errors

Turbo Pascal 3.0 had an extremely convenient system for finding the location in the source code of a runtime error, given the reported address of the error. The compiler simply recompiled the source code until it reached the offset in the object code that matched the address of the runtime error.

This system only works with the .COM code files produced by versions of Turbo Pascal prior to 4.0. Things are far more complex now, and with multiple code segments and .EXE files, the compiler needs additional information to relate the location of a runtime error in the object code to a line in the source code file.

You can insert this information into your program object code file by using the **D**ebug information item in the Compiler submenu of the **O**ptions menu. The **D**ebug information item is either **On** or **Off**; make sure that it is **On** when you compile. Generating the debug information adds nothing to the size of your file, but code speed will be reduced slightly.

Once you have set the **D**ebug information item to generate the necessary information and insert it into your file, there are two methods of locating runtime errors.

The first is when you are running your program from within the Environment. If a runtime error occurs, you will be bounced back into the Environment without a normal exit from your program The editor will load the source code (if available) for the code module in which the error occurred. In other words, if your program uses a number of units, and the runtime error occurs within one of the units, the source code for that unit will be loaded into the editor, and the cursor will be positioned as close to the runtime error location as Turbo Pascal can pin it. If the error occurred in the main program module, the main program source code will be loaded instead.

If the compiler cannot find the source code for the code module that is at fault, a dialog box will appear, entitled **Cannot find run-time error file:** The name of the source code file it wants to load will appear within the box. You then have the option of editing the file name or path within the dialog box to reflect the current location or name of the source code file. Once you press Return, the editor will attempt to load the file again.

This first method makes use of all the features of the Turbo Pascal Environment, and is nearly automatic. All you have to do is make sure that the **D**ebug information item is turned **On**, and that the source code files of all code modules you are developing are available to the Environment if it needs them.

The second method is necessary when you run your program from DOS and trigger a runtime error. An error message will appear, similar to this:

```
Runtime error 200 at 0028:02BF.
```

You need to manually write down the code and the address, because the Environment is not in memory to note it for you.

With that information written down, bring up the Environment and load the source code for your main program, regardless of where the runtime error occurred. To find the error, the Environment must begin with the main program.

Bring up the **C**ompile menu and select the **F**ind error item. A dialog box will appear, entitled **Error Address.** Type in the full address you copied from the runtime error message on the screen, and press Return. The Environment will take it from there, load the appropriate source code file as described above, and put the cursor as close to the error as it can.

Keep in mind that when using this second method, the Environment cannot tell you what kind of runtime error occurred. You have to look the error code up yourself.

The Edit File and the Primary File

The next item on the Compile menu is the **P**rimary file item. This item affects which source code file is compiled when you request a compile operation. Ordinarily, the file in the editor window (the edit file) is what you compile when you want to compile something. For simple programs that exist in only one piece, this will always be the case.

However, once you start cutting up a program source code file into several separate units, the picture becomes more complicated. You may be editing and testing a unit file, but you can't simply compile and run a unit all by itself. The unit must be linked to the main program file, and then run along with the main program.

What happens is that you are editing a unit file, but you need to compile, link, and run the entire program. You can, of course, compile the unit to disk as a .TPU file, then load and compile the main program and run that as a way of testing the unit file you're working on.

It doesn't have to be that difficult. The Environment recognizes what is called the *primary file.* The primary file is the source code file that is compiled and linked when either the **M**ake or **B**uild items in the **C**ompile menu are selected. By setting the primary file to the main program file, you can edit the unit file in the editor window, and then press F9 (the shortcut to the **M**ake item) to re-**M**ake the whole program, starting with the main program specifed in the primary file name.

Understanding that last sentence requires that you know what it means to *make* or *build* a program. This is part of Turbo Pascal's project management feature, which I'll be describing in detail in Chapter 29. I'll define the words only briefly here; you might look ahead to Chapter 29 if you're still puzzled.

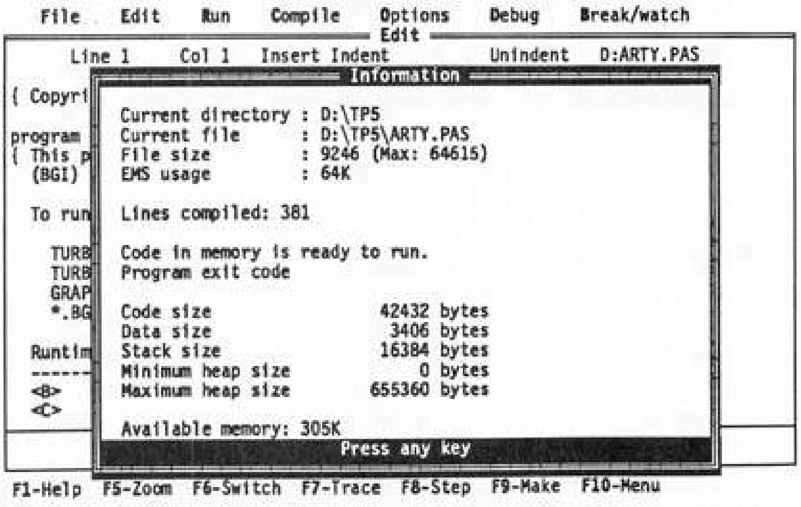
When you *make* a program file, you recompile the main program and any units that were modifed *since the last time the main program was compiled.* If none of the units were modifed, only the main program is recompiled. The units' .TPU files are *linked* with the main program, but the unit source code is not *recompiled.*

On the other hand, when you *build* a program, you recompile everything, including all units the main program uses, regardless of what was modified when.

In reality, make and build are a little more complicated than that. Again, refer to Chapter 29 for more information.

Figure 27.1

The Compile Information Box


File Edit Run Compile Options Debug Break/watch
Edit
Information
Press any key
F1-Help F5-Zoom F6-Switch F7-Trace F8-Step F9-Make F10-Menu

```
   Line 1     Col 1   Insert Indent          Unindent   D:ARTY.PAS

( Copyri   Current directory : D:\TP5
           Current file      : D:\TP5\ARTY.PAS
program    File size         : 9246 (Max: 64615)
( This p   EMS usage         : 64K
  (BGI)
           Lines compiled: 381
  To run
           Code in memory is ready to run.
   TURB    Program exit code
   TURB
   GRAP    Code size                42432 bytes
   *.BG    Data size                 3406 bytes
           Stack size               16384 bytes
 Runtim    Minimum heap size            0 bytes
 ------    Maximum heap size       655360 bytes
 <B>
 <C>       Available memory: 305K
```

Displaying Compile Information

The last item on the **Compile** menu, **Get info**, provides a summary of the various aspects of the last compile successfully completed. An example of a compile information box is given in Figure 27.1.

The box summarizes the source file size, code size, stack size, heap minimum and maximum size, and remaining system memory. It also indicates whether your compiled code resides in memory or on disk, as specified by the **Destination** item in the **Compile** menu.

If your program exits through a **Halt** statement with an exit code, that code will be reported at the bottom, along with information on any runtime error that was detected during the last time the program was run.

27.3: COMPILER DIRECTIVES

There are two ways to control the nature of the code produced by the Turbo Pascal compiler. One is through the **Options** menu, which we'll discuss in detail in Chapter 28. The other way is through commands that you place directly into your source code as you edit it. These are called *compiler directives.*

There are actually three very different types of compiler directives:

- **Switch directives** turn some condition on or off, by naming the condition and then using the "+" symbol to turn the condition on, and the "-" symbol to turn the condition off.
- **Parameter directives** provide parameters to the compiler, such as file names or values for memory allocation.
- **Condition directives** provide for the compilation of different portions of a single source file depending on parameters defined by you the programmer. This is a fairly advanced technique that touches on project management, and I will cover it in detail in Section 29.2 rather than in this section.

In virtually every case, you can set up the same compiler commands either through the menus or through compiler directives in the source code. There are several good reasons to use compiler directives rather than the **O**ptions menu. Most of the compiler directives have the advantage of being changeable during the course of program compilation. The most common example is the turning on and off of runtime error trapping. There are times when you will want the compiler to trap serious computational errors, and other times when you will prefer to be notified of them (via **IOResult**, for example) and take action on your own. The **$I**+/− compiler directive lets you turn off error trapping for a statement or two and then turn it back on again.

Then there is the more practical issue of having to load in a new configuration file every time you change source code, if you choose to depend on the menus to configure compilation. It makes much more sense to set up a series of default conditions that makes sense for the work you generally do, and then use compiler directives in your source code to set things up specifically for the project you are working on at any given time. Loading the source code loads its options for you automatically (because the options are compiler directives that are part of the source code), and saves that extra step.

Syntactically, compiler directives are special-purpose comments. If the first character in a comment is a dollar sign **$** that comment will be interpreted by the compiler as a compiler directive. *There can be no space or other characters between the left comment delimiter (either (* or {) and the dollar sign.*

Several compiler directives may be included in a single comment as a list separated by commas. In this case only the first directive need be preceded by a dollar sign. Here are some valid example compiler directives:

```
{$I-}
{$I A:MOUSELIB.SRC}
{$C-,R-}
(*$B- *)
(*$U-,V-,C-*)
```

As with all comments, you must close the compiler directive comment with the same delimiter it starts with. In other words, compiler directives like **{$I−*)** or **(*$B+}** are invalid and dangerous, since their effects are unpredictable.

Keep in mind that switch directives (those directives followed in the discussions below by +/−) are toggles which are either on + or off −. You must pick only one in practice; in other words, a directive like **$I**+/− is not valid in your source code, but is used here as a typographical device only to indicate that the **$I** switch directive has two states specified by plus and minus symbols.

$A+/−: Word alignment; default + (5.0 only)

The 86 family of processors can access data more quickly if that data is *word-aligned,* that is, if the start of data occurs at an even-numbered address. Even-numbered addresses represent what are called "word boundaries," and beginning a data item on a word boundary makes that data item word-aligned.

It may be that a data item would fall on an odd-numbered address because it follows another data item containing an odd number of bytes. In that case, if word alignment is in force, the compiler will add a single-byte "pad" to force the new data item to begin on a word boundary.

Word alignment makes for faster programs, at the cost of some (but not a great deal) of wasted space in your single data segment. The default is word alignment; if you begin to run out of data segment space, the first thing you should do is disable word alignment so that the compiler will pack data in the data segment nose-to-tail and make the best use of space, at the cost of some (but not much) code performance.

The **$A**+/− compiler directive controls word alignment. **$A+** (the default) specifies word alignment, while **$A−** disables word alignment.

This directive is equivalent to the **A**lign data item on the **C**ompiler submenu of the **O**ptions menu.

$B+/− : Boolean evaluation; Default −

There are two different ways that the compiler can generate code to support Boolean expression evaluation: Complete and short-circuit. Complete evaluation evaluates every part of an expression that strings several expressions together with AND and OR operators, even when some early part of the expression determines the outcome of the entire expression early on. Short-circuit evaluation stops evaluation of the expression as soon as the outcome of the expression is determined. This concept is discussed in detail in Section 13.2.

{$B+}** forces the compiler to generate code that performs complete Boolean expression evaluation. **{$B−} forces the compiler to generate code that performs short-circuit Boolean evaluation.

This directive is equivalent to the **B**oolean evaluation item on the **C**ompiler submenu of the **O**ptions menu.

$D+/− : Generate debug information; default −

In order to locate the position of a runtime error in your source code, the compiler needs more information than simply the address at which the error occurred (as was sufficient for Turbo Pascal 3.0). When directed to with the **{$D+}** directive, the compiler creates a table of line number addresses for each procedure in the program. This table allows the Environment to use an address provided by you to "zero in" on a specific line number in your source code.

When compiling a unit, the debug information is recorded in the unit's .TPU file. When compiling a program into memory, this informaton is retained in memory. When compiling a program to disk, the information is written out to a separate .TPM file, *if* the generation of .TPM files is enabled with the **{$D+}** directive.

If the generation of debug information is disabled by the **{$D−}** directive (the default condition) the Environment will be unable to pinpoint the location of a runtime error for you.

If you're using Turbo Pascal 5.0, the matter of debug information will be taken up again in Chapter 30.

$E+/− : Floating point emulation; default + (Version 5.0)

Version 5.0 of the compiler has the ability, shared by Turbo C, of emulating the '87 family of math coprocessors if one is not present in the system. This allows you to make use of the IEEE floating point types **Single**, **Double**, **Extended**, and **Comp**, even if you don't have an '87 installed. Turbo Pascal 4.0 users need to have the actual chip in their systems to use these types.

The default is to have emulation enabled (**{$E+}**). If you turn emulation off, you will need to have an '87 to use the IEEE floating point types, but your code will be somewhat more compact and considerably faster.

$F+/− : Force far calls; default −

The code that calls procedures or functions can be generated two ways: If the call must venture outside the current 64K code segment, the call is a *far* call; otherwise, the call is limited to the current 64K code segment and is a *near* call. The main program and its subprograms are contained within a single code segment, so the subprograms are accessed using near calls. The subprograms defined within a unit, however, must be called as far calls because they can be called from the main program's code segment or any of the other unit code segments.

In most cases, the compiler decides whether to generate a subprogram using near or far calls. However, there are certain situations in which a subprogram within the main program's segment must be compiled for far calls. The main program's exit procedure is the most common example, because it is joined into a chain with all unit exit procedures that spans several code segments (see the discussion of exit procedures in Section 17.4).

All subprograms compiled after a **{$F+}** directive are compiled for far calls. It's not enough simply to turn far calls on, however, since typically only one or two subprograms within a program need to be compiled for far calls. The correct way to use **$F** is to bracket a far procedure or function with **{$F+}** and **{$F−}**:

```
{$F+}

PROCEDURE MyFarProc;

BEGIN
  ...
END;

{$F-}
```

This way, only the single procedure becomes a far procedure, which is as you want it. Far procedures and functions take up more room and require more time to execute because of the extra burden of pushing both a segment address and an offset address on the stack, rather than just an offset address.

For this reason, it is not a good idea to enable far calls through the equivalent Force far calls item on the **C**ompiler submenu of the **O**ptions menu. Using the menu makes *all* calls in a program far calls, indiscriminately. Use a razor, not a club.

However, Version 5.0 overlays require that far calls be used *throughout* a call chain that ends with a procedure within an overlay. In other words, if procedure **A** calls procedure **B**, procedure **B** calls procedure **C**, procedure **C** calls procedure **D**, and **D** is within an overlay, then **A**, **B**, **C**, and **D** must *all* be compiled as far. The safest way to anticipate all combinations is to simply use far calls for everything in an overlaid application. For overlays, use the club; the razor will kill you.

$I+/− : I/O error trapping; default +

This toggle determines whether I/O errors are trapped or simply reported. With **{$I+}** in force (the default) the Turbo Pascal runtime code traps I/O errors with an error message and then brings the program to a halt. With **{$I−}** in force, trapping is turned off. Errors will still be *reported* by the **IOResult** function (see Section 19.4) but it will be up to you to test **IOResult** for errors and take appropriate action when errors are reported.

As mentioned above in connection with the **$F** directive, don't just turn trapping off and leave it off. I/O trapping is there for a reason, and it can be very helpful in tracing a problem when an I/O error occurs. It's good practice to turn trapping off only around the single statement which would generate a predictable, trappable I/O error; for example, **Reset** or **Rewrite**:

```
Assign(MyFile,'FOO.TXT');
{$I-} Reset(MyFile); {$I+}
IF Ioresult <> 0 THEN CallFileErrorProc;
```

Trapping should be left on at all other times to capture the unexpected.

As with **$F**, you can turn off I/O trapping through the **I**/O checking item on the **C**ompiler submenu of the **O**ptions menu, but again, *using the menus with this option is* *a bad idea.* Turn off error trapping only when you must, leaving as little of your code as possible exposed to unreported I/O errors.

$I <filename> : Include file

I think it was a bad idea to have two different forms of the **$I** directive for two different functions, so take special note that **$I <filename>** has *nothing* to do with **$I+/−** as described above.

$I <filename>** is a parameter directive that allows you to "include" a file during compilation. This is a way to cut your source code up into chunks. When the compiler encounters the **$I <filename> directive, it opens the disk file **<filename>** and begins compiling it. The file **<filename>** is not actually read into memory as a whole; the compiler simply reads it line by line, compiles the line to machine code, then throws the line away and reads the next line.

Note that starting with Turbo Pascal 4.0, include files *may* include any other files, up to eight nested levels deep. One new restriction on **$I <filename>** is that it may not appear between any **BEGIN..END** block. Turbo Pascal 3.0 and earlier versions allowed include files to be included anywhere within a source file.

Using include files was once the only way to divide a large Turbo Pascal program up into manageable chunks. Many programmers created libraries of procedures and functions that were include files, and then used **$I <filename>** to "link" these libraries into their main programs. This system will still work, but I strongly encourage you to place your libraries of utility functions into separately compiled units. Not only does this save the time spent compiling trusty, proven routines each time you compile the main program, but it also takes your utility libraries out of your main program's code segment. Once your program begins to require more than 64K of code, this will be your only way out. Use units, not include files, for subprogram libraries!

$L+/− : Enable in-memory link buffering; default + (4.0 only!)

This switch directive controls a speed/memory tradeoff that may become important once your programs get ambitious. The Turbo Pascal linker reads each .TPU file named in a **USES** statement twice. With **{$L+}** in force (the default) the .TPU files are buffered in memory between the two passes. With **{$L−}** in force, the .TPU files are reread from disk for the second pass, which takes more time but frees more memory.

You won't need to change this switch unless you're working on a *very* large program. Still, if you begin to have trouble compiling within available memory, switching to **{$L−}** is the first thing to try.

This directive is equivalent to the Link buffer item on the Compiler submenu in the Options menu.

Version 5.0 *completely* redefines this directive; see below:

$L+/− : Enable local symbol generation; default + (5.0 only!)

To use the Integrated Debugger with programs that contain subprograms (i.e., 98% of all sensible programs), you must explicitly enable the generation of a table of local symbols, that is, symbols belonging to the subprogram rather than the main program. Disabling local symbol generation releases a little memory but has no other real consequence. The default is to generate local symbols and there is little reason to disable it.

$L <filename> : Link object file

This parameter directive specifies the name of an external machine-code file to the Turbo Pascal linker. The linker then loads the file into memory and links it into the generated code when the external routine is called.

For example, if you have an external file called GAMER.OBJ and wish to link it into your program, you would use this syntax:

```
{$L GAMER.OBJ}
PROCEDURE Stick(Sticknumber : Integer; VAR X,Y : Integer);
               EXTERNAL;
```

This assumes that a file named GAMER.OBJ resides in the current directory. You could specify a file in a different directory by including directory path information in the **$L** directive. Alternatively, you can store a directory name in the Include directories item on the Directories submenu of the Options menu. The Turbo Pascal compiler will search there for an external file if no path is given within the **$L** directive, and the file is not found in the default directory.

$L assumes a .OBJ extension. Specifying GAMER is equivalent to specifying GAMER.OBJ. If you wish to link an external file with some other extension, you must specify the extension explicitly.

Keep in mind that if you specify a file name in a **$L** directive and then fail to make use of the file through an **EXTERNAL** declaration later in the program, Turbo Pascal will generate this error:

```
Error 51: Invalid PUBLIC definition (GAMER)
```

after it reaches the end of the file.

$M <stacksize>, <heapmin>, <heapmax> : Memory allocation; default {$M 16384,0,655360}

You have the power to alter the allocation of memory to your program. Specifically, you can specify how much memory is to be used for the stack segment, and how much is to be made available to the heap for storage of dynamic variables. The way you specify allocation values is through the **$M** parameter directive.

The **$M** directive can be used *only* within program files. If you place one in a unit source file, no error will be generated, but the directive will be ignored and will have no effect. **$M** takes three numeric parameters: Stack allocation, minimum heap allocation, and maximum heap allocation. The three values are placed within the comment delimeters, separated by a space from the **$M** directive name, and separated from one another by commas.

For example, in the following directive:

```
{$M 65520,0,32768}
```

the stack allocation is 65,520 bytes; the minimum heap allocation is 0 bytes, and the maximum heap allocation is 32,768 bytes.

Stack allocation is the amount of memory in your program's stack segment. Turbo Pascal uses the stack heavily, for keeping temporary values, and especially for the passing of values to and from subprograms. Therefore, you must have some minimum amount of memory available for stack use, or your programs will not run at all.

The minimum amount of stack that you can allocate to a Turbo Pascal program is 1024 bytes. If you try to allocate less than that, you will receive an error:

```
Error 17: Invalid compiler directive
```

The maximum amount of memory allocatable to the stack is 65,520 bytes, *not* 65,536 bytes (what we often call "64K"). The default stack allocation (that is, the allocation granted automatically unless you override it with a **$M** directive) is 16,384 bytes. This is a good working value, and should be left alone unless you really need to change it.

More stack is often required when your program makes heavy use of recursion (see Section 14.3). Every time a recursive procedure calls itself, it allocates a new "stack frame" on the stack with new instantiations of all parameters and local variables. Make a few hundred recursive calls, and this adds up...which you must keep in mind if you intend to build recursion into your programs. If you run out of stack space, you will generate runtime error 202, *if* stack overflow checking is enabled. (See the **$S+/−** directive, on page 737.) If stack checking is not enabled, your program will crash mutely, leaving you scratching your head and reaching for the power switch.

Minimum heap allocation is the minimum amount of memory that your program must be able to allocate to the heap, or it will refuse to run. This is a "guarantee" that your program, once it runs, will have a certain amount of RAM available for essential buffers and variables on the heap. The default value is 0; that is, your program can run by default without any memory for the heap at all. If you make *any* use of the heap, decide how much memory your program needs to get its work done, and make this the minimum heap allocation value.

Maximum heap allocation is the amount of memory that your program should reserve for the heap, if the memory is available. The default value is 655,360 bytes, which (if you know your PC architecture) is the entire useable RAM area for PC DOS applications.

The default values for minimum and maximum heap allocation means that, by default, your PC will *promise* you *no* heap memory, but will *take* all the memory it can find, right up to the end of DOS memory.

If you intend to make use of **Exec** in the **DOS** unit to spawn child processes, you *must* cut back on the maximum heap allocation by the amount of memory required by your child process. I discuss this use of **$M** pretty thoroughly in Section 20.10, in connection with the **Exec** procedure.

You can set memory allocation through the **M**emory sizes item on the **C**ompiler submenu of the **O**ptions menu, but again, I don't think that's a very good idea. Use the defaults when you can, and override them when you must on a program by program basis, by using the **$M** directive.

$N+/− : Numeric processing; default −

There are two broad categories of real numbers available within Turbo Pascal. One is the "software-only" real numbers, of the predefined type **Real**, which occupy six bytes. The other is the "8087-style" real number types: **Single**, **Double**, **Extended**, and **Comp**.

In Turbo Pascal 4.0, the 8087 real number types *require* a math coprocessor like the 8087, 80287, or 80387 to compile and run. The Turbo Pascal compiler can detect the presense or absense of a math coprocessor on your system, and if it fails to find a math coprocessor it will not let you use the 8087 types.

Floating point support for Turbo Pascal 5.0, on the other hand, resembles that of Turbo C in that the math coprocessor is used if available, and emulated if not. You needn't be concerned whether someone running your compiled program has a math coprocessor or not, except that emulating a coprocessor is considerably slower than using a real one. Version 5.0 emulation may be turned on and off through the **$E+/−** directive, as explained above.

The **$N** switch allows or forbids the use of the 8087 types. The default condition is **{$N−}**, meaning that you cannot use the 8087 types. Inserting a **{$N+}** directive into your program allows you to compile a program using the 8087 types. Keep in mind that under Version 4.0, if you don't have an 8087 (or one of the more advanced Intel math coprocessors) installed, *the compiler will not let you use the 8087 types, whether {$N+} is in force or not!*

This directive is equivalent to the **N**umeric processing item on the **C**ompiler submenu of the **O**ptions menu. The "hardware" state on the menu is equivalent to **{$N+}**, while the "software" state is equivalent to **{$N−}**.

$O+/− : Enable unit overlay compilation; default − (5.0 only!)

For a unit to be compiled as an overlay under Turbo Pascal 5.0, the unit must contain a **{$O+}** directive enabling overlay compilation of the unit. When enabled, the directive allows the unit containing it to be compiled to the .OVR overlay file rather than the main program's .EXE file. Note that including a {O+} directive in a unit does not *force* a unit to be overlaid; it merely *allows* the unit to be overlaid. To be overlaid, a unit must contain the **{$O+}** directive, and the main program must name the unit in a **$O <unit name>** directive.

$O <unit name>: Specify overlay unit name (5.0 only!)

Every unit to be used as an overlay must be named as an overlay in the main program source file, within a **{$O}** directive. Each unit must have its own separate directive, existing *after* the **USES** statement but before anything else:

```
PROGRAM JiveTalk;

{$F+}

USES Overlay,DOS,CRT,CircBuff,XMODEM,Kermit,Packet1K,Parser;

{$O XMODEM}
{$O Kermit}
{$O Packet1K}
```

Also, each unit to be overlaid must contain a {$O+} directive, as explained above.

$R+/– : Index range error check; default –

Ordinarily, if you attempt to index an array or string outside of its legal bounds, no runtime error occurs. If indexing outside the bounds of an array or string trashes adjacent data items, the resulting bugs will be peculiar and rough to define. You have the option of turning on runtime index range error trapping, so that when an index range error happens, the Turbo Pascal runtime code will trap a runtime error and halt your program. The **$R+/–** directive turns this error trapping for index range errors on and off. **{$R+}** turns index range error trapping on. **{$R–}** (the default) turns trapping off.

Like all of Pascal's safeguards, range checking costs your program a little in terms of execution speed, and makes the generated code a little larger. For ease of debugging you might turn index range trapping on while you're developing a program. Once you have thoroughly debugged your program, you might wish to recompile it with trapping turned off. Your program will then run a little faster.

This directive is equivalent to the **R**ange checking item on the **C**ompiler submenu of the **O**ptions menu.

$S+/– : Stack overflow checking; default +

Ordinarily, the Turbo Pascal runtime code checks to see if space is available on the stack for local variables and value parameters before each call to a function or procedure. If space is not available, a runtime error will occur and execution will cease.

With **{$S+}** (the default) in force, the compiler will generate code to make this check. With **{$S–}** in effect, that code will not be generated, and the runtime library will simply assume that space is always available on the stack. This will usually be true. However, in recursive applications where heavy use of the stack is expected, it may be a good idea to use **{$S+}**.

With **{$S−}** in force, a stack collision will almost always crash your system hard, requiring a reboot.

This directive is equivalent to the **S**tack checking item on the **C**ompiler submenu of the **O**ptions menu. It also directly replaces the **$K+/−** directive from Version 3.0 and earlier.

$T+/− : .TPM file generation; default − (4.0 only!)

This switch directive determines whether a .TPM (Turbo Pascal Map) file is generated when the program is compiled to a disk file. A .TPM file contains names of variables and subprograms and other information about where they are located in a file. This information may be used by symbolic debuggers such as Periscope.

When **{$T−}** (the default) is in force, no .TPM file is generated. If **{$ST+}** is in force, a .TPM file is generated. but *only* for a main program (.PAS) file compiled *to disk*. Units, and programs compiled to memory, are not affected and will not generate a .TPM file regardless of the state of $T.

$T is related to **$D**, which controls the generation of debug information for locating runtime errors. If you're going to generate a .TPM file for use by a symbolic debugger, be sure to include a **{$D+}** directive in your program, and also in any of your units that the program uses. If all you're going to do is have Turbo Pascal locate a runtime error for you, you need only use the **{$D+}** directive. *{$T+} is not necessary to find a runtime error location.*

Note that **$T+/−** is *not* supported under Turbo Pascal 5.0! Map files for symbolic debugging are unnecessary in Version 5.0. High-level debugging is something built-in to Turbo Pascal 5.0; for the whole story refer to Chapter 30.

This directive is equivalent to the **T**urbo Pascal map file item on the **C**ompiler submenu of the **O**ptions menu.

$U <filename> : Unit file name; no default (4.0 only!)

In most cases, the name of a unit, the name of the unit's source file, and the name of the unit's .TPU file are one and the same. Deviations from this rule must be specified using the **$U** parameter directive within the **USES** statement of a program or unit using the unit in question.

This is best explained by example. If you create a unit named **DotPlot**, it will typically have a source file named DOTPLOT.PAS, and a .TPU file named DOT-PLOT.TPU. Your programs could use this unit by including its name in their **USES** statements:

```
USES DOS, Crt, DotPlot;
```

However, suppose you would like to call your unit **HPPlotterSupport**, for readability's sake. This is too long for a DOS file name, so you could rename the unit within the unit's header, but keep the source and .TPU files named DOTPLOT. Then, to use

HPPlotterSupport, you would need to specify the name of the .TPU file containing **HPPlotterSupport** in a **$U** directive:

```
USES DOS, Crt, {$U DOTPLOT} HPPlotterSupport;
```

This statement tells the Turbo Pascal compiler that unit **HPPlotterSupport** is to be found in a file named DOTPLOT.TPU.

If you intend to create units with file extensions *other* than .TPU, you must specify the nonstandard extension within a **$U** directive:

```
USES DOS, Crt, {$U DOTPLOT.U} HPPlotterSupport;
```

Important note: The **$U <filename>** directive has been dropped from Version 5.0.

$V+/− : String VAR-parameter length checking; default +

All string types have a specific physical length. The default length of type String without a length qualifier is 255 bytes (see Section 9.5). String types of shorter lengths are defined as **String[20]**, **String[80]** and so on, with the length given in brackets after the identifier **String**. When you pass a string type to a function or procedure by reference (as a **VAR** parameter, in other words) the physical length of the formal parameter and the physical length of the actual parameter are required to be exactly the same. Attempting to plug a larger or smaller actual parameter into the formal parameter will generate a compile-time error.

This restriction makes it awkward to write a procedure to handle string types of differing physical sizes. The **$V** directive allows you to turn this restriction off and on. **{$V+}** (the default) enforces this physical length matching restriction. **{$V−}** allows you to pass actual parameters of any length to a given formal parameter.

The consequences of passing an actual parameter *larger* than its formal parameter can be the overwriting of adjacent data, very much like array index range errors.

This directive is equivalent to the Var string checking item on the Compiler submenu of the Options menu. The "strict" state from the menu is equivalent to **{$V+}**, and the "relaxed" state is equivalent to **{$V−}**.

27.4: THE COMMAND-LINE COMPILER

As part of the Turbo Pascal product you also get (in addition to the Turbo Pascal Environment) a "naked" version of the Turbo Pascal compiler that does not provide an editor or any menu-driven functions. This compiler is invoked from the DOS command line, and all commands and directives are passed to it as DOS command-line parameters.

People whose first experience with compilers was with the earlier Turbo Pascal Environment may wonder why anyone would want to use the standalone command-line compiler introduced with Turbo Pascal 4.0.

The command-line compiler is a classic DOS *filter*. It takes one file, your source file, as its input file, and creates a second file, the object code file, as its output. While working, it does nothing visible aside from displaying the name and line number of the file it is currently processing. No windows, no editor, no menus. Why use it indeed?

Actually, there are two general reasons people use the command line compiler:

1. Some programmers are very attached to the text editor they use in other work, and don't want to learn yet another set of editing commands. Most of the time you actually spend programming is spent working in a text editor, and editors, like music and beer, are matters of intimate personal taste. With Turbo Pascal's command-line compiler, you can use whatever editor you wish, and treat Turbo Pascal the same as any other command-line driven utility.
2. Perhaps more important, some very large, complicated programming projects are more readily managed from the command line, using the MAKE.EXE utility. If your application comes to span 100,000 lines of Pascal and assembler code and takes half an hour to rebuild, you might as well let the command-line compiler handle it, *especially* if you are using a 386 machine that allows you to run the command-line compiler as a background task.

Running TPC.EXE

The name of the command-line compiler is TPC.EXE. You run it from the command-line:

```
C:\>TPC [OPTIONS] FILENAME [OPTIONS]
```

The name of the source file may be preceded or followed (or both) by one or more command-line options, which give the compiler additional instructions or override compiler defaults. For example, the following is a typical invocation of TPC:

```
C:\>TPC /$B+ MYPROG /$M16384,0,4000 /B
```

If you don't provide a file extention to the file name, TPC will assume .PAS. To compile a file with an extension of anything *other* than .PAS, you must include the extension.

Invoking TPC without a filename brings up a summary of its invocation syntax and all the legal command-line options:

```
Turbo Pascal  Version 4.0  Copyright (c) 1987 Borland International
Syntax: TPC [options] filename [options]
/B       Build all units            /$B+    Complete boolean evaluation
/Dxxx    Define conditionals        /$D-    No debug information
/Exxx    Executable directory       /$F+    Force FAR calls
```

```
/Fxxx   Find run-time error      /$I-     No I/O checking
/Ixxx   Include directories      /$L-     No link buffer
/M      Make modified units      /$Mxxx   Memory allocation parameters
/Oxxx   Object directories       /$N+     8087 numeric co-processor
/Q      Quiet compile            /$R+     Range checking
/Rxxx   Run in memory            /$S-     No stack checking
/Txxx   Turbo directories        /$T+     Generate TPM file
/Uxxx   Unit directories         /$V-     No var-string checking
/Xxxx   Execute on disk
```

This is the equivalent of a help system for TPC. The example shown here is specifically from Turbo Pascal 4.0; the 5.0 display is very similar. Keep it in mind; it can save you some time running for the manual when you don't quite remember which switch does what.

Compiler Mode Options

There are two kinds of options, and the way to tell them apart (as you can see from the "help" summary above) is that some are preceded by dollar signs, and some are not. Those without dollar signs are called *compiler mode options,* and give instructions to the compiler about the compile operation underway. The second kind of option parallels the compiler directives you build into your source code with the **$I** compiler directive, hence the dollar sign. We'll deal with compiler mode options first.

/B (Build)

/B instructs TPC to rebuild the entire program, including all its unit files, regardless of the modification dates of any source files involved. This is identical to the **B**uild item in the **C**ompile menu, which I'll discuss in connection with project management in Section 29.1.

/D (Define Conditionals)

Conditional compilation allows you to embed blocks of source code in your programs that either compile or do not compile depending on the presence or absense of certain controlling symbols that you define. You might have two different versions of a highly optimized matrix inversion procedure; one that makes use of the 8087 math coprocessor, and the other that makes use of Turbo Pascal's 6-byte software-only real number type. Both versions of the procedure cannot be compiled at once, since they both have the same name. Instead, the compiler chooses one or the other to compile depending on whether a certain symbol is defined or not defined.

You can use the **/D** directive to define a symbol for use in conditional compilation. Simply follow the **/D** directive *immediately* with the symbol. *Do not insert an intervening space:*

```
C:\>TPC JLIST2 /DCANON
```

Here, the symbol **CANON** is defined using **/D**. This allows you to conditionally compile a block of code based on the symbol **CANON:**

```
{$IFDEF CANON}

<code here compiled ONLY if symbol CANON is defined>

{$ENDIF}
```

The code that falls between the **$IFDEF** and the **$ENDIF** directives will compile *only* if the symbol CANON has been defined. Otherwise, TPC ignores it.

For the full story on conditional compilation, see Section 29.2

/GS, /GP, /GD (Generate Map File Options.) (5.0 Only!)

The 4.0-style .TPM files are no longer produced by Version 5.0. Instead, more detailed map file information can be specified by the three directives shown here:

/GS : generate segment information only
/GP : generate publics information only
/GD : generate detailed information (everything)

There is no source code compiler directive equivalent to any of these command-line directives. There are equivalents, however, in the 5.0 Link item in the **Options** menu.

/M (Make)

The **/M** directive tells TPC to perform a Make operation on the source file being compiled. This does *not* involve the stand-alone MAKE.EXE utility, but instead mimics some of MAKE's simpler capabilities. I'll discuss the compiler's built-in Make feature in more detail in the Project Management section, Chapter 29.

In brief, the Make feature takes a look "back in time" to see if any units used by the compiland (the source code being compiled) were modified *since* the compiland. If so, the compiler compiles those units as well. Furthermore, if a unit being recompiled uses any other units, those units are checked as well. The compiler, in effect, follows this "dependency chain" back as far as it goes to ensure that all .TPU files in the current directory are up to date with their source files. If any are not, they are recompiled.

This option is identical to the **M**ake item on the **C**ompile menu. Again, for more details on the Make feature, see Section 29.1.

/V (Turbo Debugger .EXE Information) (5.0 only!)

The /V directive builds debug info tables into the .EXE file for use by the standalone Turbo Debugger symbolic debugger. It is equivalent to the **S**tandalone debugging item on the 5.0 **D**ebug menu.

Compiler Directive Options

TPC supports all of the compiler directives described in Section 27.3. All of the default conditions are the same, and they are passed to TPC in almost the same way: By placing the directive (including its dollar sign symbol) on the command-line, preceded by a slash.

In other words, to compile MYPROG.PAS with TPC with complete Boolean evaluation and numeric processing done in hardware, you would invoke TPC with the following command-line:

```
C:\>TPC MYPROG /$B+ /$N+
```

Each option must have its own slash and its own dollar sign.

Since I've covered all the directive options in detail elsewhere, I won't describe them again. Table 27.1 summarizes the options and cross-references them to both the source code directives described in Section 27.3 and the **O**ptions/**C**ompiler menu items summarized in Section 28.1.

The **/$M** directive takes its three parameters immediately after the M, separated by commas:

```
C:\>TPC MYPROG /$M32768,4096,655360
```

Directory Options

TPC recognizes several directives that are equivalent to the various items in the **D**irectories submenu of the **O**ptions menu. These directives specify directory paths for include files, .TPU files, etc.

Table 27.1
Compiler Options and Directives Cross Reference

Options/Compiler Item	**Equivalent Directive**		*Discussed*	
** = Default State*	*Source Code*	*TPC*	*on Page Number*	
*Range checking Off	$R−	/$R−	737	
Range checking On	$R+	/$R+	737	
Stack checking Off	$S−	/$S−	737	
*Stack checking On	$S+	/$S+	737	
I/O checking Off	$I−	/$I−	732	
*I/O checking On	$I+	/$I+	732	
Debug information Off	$D−	/$D−	730	
*Debug information On	$D+	/$D+	730	
*Turbo Pascal map file Off	$T−	/$T−	738	
Turbo Pascal map file on	$T+	/$T+	738	
*Force far calls Off	$F−	/$F−	731	
Force far calls On	$F+	/$F+	731	
*Overlays allowed Off	$O−	/$O−	736	V5.0
Overlays allowed On	$O+	/$O+	736	V5.0
*Align data Byte	$A−	/$A−	730	V5.0
Align data Word	$A+	/$A+	730	V5.0
Var string checking Relaxed	$V−	/$V−	739	
*Var string checking Strict	$V+	/$V+	739	
*Boolean evaluation Short circuit	$B−	/$B−	730	
Boolean evaluation Complete	$B+	/$B+	730	
*Numeric processing Software	$N−	/$N−	736	
Numeric processing Hardware	$N+	/$N+	736	
Emulation Off	$E−	/$E−	731	V5.0
*Emulation On	$E+	/$E+	731	V5.0
Link buffer Disk	$L−	/$L−	733	V4.0
*Link buffer Memory	$L+	/$L+	733	V4.0
Local symbols Off	$L−	/$L−	733	V5.0
*Local symbols On	$L+	/$L+	733	V5.0
Conditional defines (no default)	$DEFINE	/D	770	
Memory sizes *(16384,0,655360)	$M	/$M	414,734	

/E (Executables Directory)

.EXE files created by TPC are ordinarily written into the default directory, but you can supply the name of another directory for executable files with the **/E** directive. Specify the full pathname *immediately* after the /**E** directive. Leave no intervening space! The example below directs TPC to write its .EXE file into the directory E::

```
C:\>TPC JLIST2 /ED:\PRINTER
```

The /**E** directive is identical to the **E**xecutable directory item on the **D**irectories submenu of the **O**ptions menu.

/I (Include Directories)

Ordinarily, TPC searches only the current directory for files included to a source file with the **$I** directive. You can specify one or more additional directories that TPC is to search *after* it searches the current directory, using the **/I** directive:

```
C:\>TPC JLIST2 /ID:\TOOLKIT;D:\HACKS
```

If you specify more than one additional directory with **/I**, separate them with semicolons *only*. Leave no intervending space characters.

The **/I** directive is identical to the **I**nclude directories item on the **D**irectories submenu of the **O**ptions menu.

/O (Object Code Directories)

As with most directories, TPC will default to searching the current directory for assembly language .OBJ files when it encounters a **$L <FILENAME>** directive in your source code. You can specify one or more additional directories to search *after* it searches the current directory by using the **/O** directive:

```
C:\>TPC JLIST8 /OD:\MASM;D:\ASMKIT
```

As with **/I**, separate multiple paths with semicolons but do not leave any intervening space characters.

The **/O** directive is identical to the **O**bject directories item on the **D**irectories submenu of the **O**ptions menu.

/T (Turbo Directory)

TPC ordinarily searches the current directory and the directory where its own .EXE file is stored when it needs to locate the TURBO.TPL and TURBO.CFG files. You have the option of specifying *one* other directory as the Turbo directory, by using the **/T** directive.

This directive is identical to the **T**urbo directory item on the **D**irectories submenu of the **O**ptions menu.

/U (Unit Directories)

The first place TPC looks for a unit that you use in compiling a file is in TURBO.TPL. TURBO.TPL is generally kept in the same directory with TPC, although you can place it elsewhere with the **/T** directive. (See above.) If a unit is not located in TURBO.TPL, TPC next checks the current directory. Finally, it will search any directories that you specify with the **/U** option.

/U allows you to pass one or more directories to TPC, separated by semicolons:

```
C:\>TPC JLIST8 /UD:\TOOLKIT;D:\TURBO\HACKS
```

The **/U** directive is identical to the **U**nit directories item on the **D**irectories submenu of the **O**ptions menu.

Program Execution Options (4.0 only!)

The last two options I'll discuss are unique to the command-line compiler TPC. Both specify a method of running the program compiled by TPC immediately after compilation without any further input from you.

/R instructs TPC to compile and link your program file into memory, then run the memory-resident program. No code is written to disk. Once the program ceases execution, control returns to the DOS command line.

/X instructs TPC to compile and link your program to a .EXE file, and then immediately load and execute the .EXE file once compilation and linking are complete.

Neither **/R** nor **/X** will run your program if any compile-time errors are discovered during compilations. Nor will either option execute a unit file. If you try, you will see:

```
Error 110: Cannot run a unit.
```

You can pass command-line parameters to your program by appending the parameter text to either the **/R** or **/X** option, enclosed in double quotes:

```
C:\>TPC MYPROG /R"INFILE.TXT OUTFILE.BIN"
```

Actually, you don't need to enclose the parameter text in quotes, but the quotes help prevent TPC from getting confused if you have slashes in your parameter text. Without the quotes, TPC might interpret a slash as the beginning of the next option.

Important note: Both **/R** and **/X** have been dropped from Version 5.0! If you wish to run in memory under 5.0, you have to use the Environment.

Setting Up a Configuration File for TPC

If you find that you're using a set of defaults different from those built into the compiler, you might want to set up a configuration file for TPC so that you don't have to pass TPC your defaults as a list of options each and every time you compile something.

When TPC is invoked, it attempts to open a file called TPC.CFG in the current directory. If TPC.CFG is found, it will be opened and its contents interpreted as a list of options. These options will be parsed and noted *before* the command line passed to TPC is parsed. This means that you can override an option set in TPC.CFG with an option on the command line if you so choose.

If no TPC.CFG file is found in the current directory, TPC will search its own directory (the one where TPC.EXE is kept) for TPC.CFG.
If you want to force TPC to look in other directories for its .CFG file, you must specify which directories using the **/T** option as the *first* option passed to TPC on the command-line.

Each line in TPC.CFG should contain a separate option. An option line may begin with a slash or a dash. If a line does *not* begin with a slash or a dash, TPC will try to interpret the line as the name of a Pascal source file to compile. Note that putting multiple source code file names in TPC.CFG will *not* force TPC to compile multiple files at one invocation; rather, only the *last* file specified will be compiled.

As an example, consider the following text stored in TPC.CFG:

```
LOCATE.SRC
/$B+
/$R-
/$N+
/$M32768,0,4096
```

If this file is located where TPC can find it, invoking TPC *without* a command line of any kind will not bring up the help summary, but instead will compile LOCATE.SRC as though you had passed the following command line to TPC:

```
C:\>TPC /$B+ /$R- /$N+ /$M32768,0,4096 LOCATE.SRC
```

28 Options and Configuring the Environment

Turbo Pascal is as powerful as it is in part because it is so configurable. There are a great many *options* that may be set, both from within the Environment and from special commands inside your source code. In general, these options affect Turbo Pascal's activities in two ways: How the compiler generates your object code file, and how the Environment looks and operates.

Configuration of both the compiler and the Environment may be done from the **O**ptions menu. Bringing up the **O**ptions menu may be done by moving the highlight over the word **O**ptions on the menu bar at the top of the screen, or else pressing Alt-O from anywhere within the Turbo Pascal Environment.

28.1: THE COMPILER SUBMENU

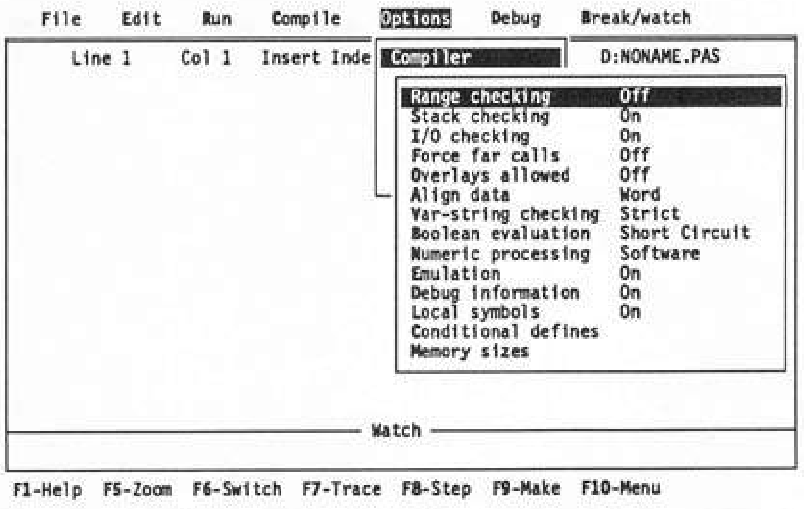
The topmost item in the **O**ptions menu is itself a menu, named Compiler. When you select Compiler, you bring up a sizeable menu of all the various options that control how the Turbo Pascal compiler generates its object code. This menu is shown in Figure 28.1.

Use Source-Code Compiler Directives!

I will risk being a curmudgeon here and state flatly that I don't think using the Compiler submenu to set compiler options is a good idea. My reason is fairly simple: Compiler

Figure 28.1

The Compiler Submenu



```
   File    Edit    Run    Compile    Options    Debug    Break/watch

      Line 1     Col 1    Insert Inde  Compiler          D:NONAME.PAS

                                        Range checking      Off
                                        Stack checking      On
                                        I/O checking        On
                                        Force far calls     Off
                                        Overlays allowed    Off
                                        Align data          Word
                                        Var-string checking Strict
                                        Boolean evaluation  Short Circuit
                                        Numeric processing  Software
                                        Emulation           On
                                        Debug information   On
                                        Local symbols       On
                                        Conditional defines
                                        Memory sizes

                                 Watch

F1-Help  F5-Zoom  F6-Switch  F7-Trace  F8-Step  F9-Make  F10-Menu
```

options are used to customize object code generation for the needs of a given program. That being the case, *the program source code should set the options to meet its own needs.* For example, if you are writing a program named **Foo** that needs lots of stack but no heap, you might set memory allocation through the Compiler submenu to give **Foo** 65,520 bytes of stack but zero bytes for heap minimum and maximum.

All well and good, but when you get tired of working on program **Foo** and load in program **Bar** to work on for awhile, the options you set for program **Foo** will still be in force. If you forget to reset the options to meet the needs of program **Bar**, **Bar** may begin working very strangely for reasons that won't be immediately obvious.

The burden of making menu-set options match the work in progress falls on you. It's far wiser to let each individual source code file tell the compiler what it needs. The way to do this is by using the various compiler directives that may be embedded in your source code. I described these in considerable detail in Section 27.3 and will not describe them again here. Instead, Table 28.1 cross-references the various compiler options available in the Compiler menu to their equivalent compiler directives, along with the page on which they are described.

The column marked "TPC" cross-references the equivalent command-line directives for the command-line version of the Turbo Pascal compiler, TPC.EXE. TPC.EXE is discussed in Section 27.4.

All but the last two items on the Compiler submenu are two-way switches; that is, the menu item can be in one of two states. You *toggle* the item between its two states by pressing the highlighted letter for a given item. For example, the default state for the Boolean evaluation item is Short circuit. To change from Short circuit to Complete, (or to flip the switch the other way) simply press the B key. Alternately, you can move the highlight bar until it covers the option you want to change, and then press Return. Pressing the highlighted letter of your chosen item is easier, because when you press the letter, the highlight bar *immediately* moves over the selected item.

The last two items on the Compiler submenu are not switches, but require actual values that you must type in response to a dialog box. The Conditional defines item brings up this dialog box directly; simply enter any condition defines that you wish to be in force for subsequent compiles.

The Memory sizes item, by contrast, brings up its own submenu with the three memory parameters on it: Stack size, Low heap limit, and High heap limit. The current values are displayed on the menu. To change one of the values, you must either press the highlighted letter of the value you wish to change, or else move the highlight bar over that item and press return. In either case, you'll bring up a dialog box into which you can type the new value for the selected item.

The Linker Submenu (5.0 Only!)

In Turbo Pascal 5.0, there is an additional submenu beneath Compiler but before Environment. This is the Linker submenu, and it specifies various options for the Turbo Pascal smart linker. The Linker submenu has only two items.

Table 28.1
Compiler Options and Directives Cross Reference

Options/Compiler Item	**Equivalent Directive**		*Discussed*
** = Default State*	*Source Code*	*TPC*	*on Page Number*
*Range checking Off	$R−	/$R−	737
Range checking On	$R+	/$R+	737
Stack checking Off	$S−	/$S−	737
*Stack checking On	$S+	/$S+	737
I/O checking Off	$I−	/$I−	732
*I/O checking On	$I+	/$I+	732
Debug information Off	$D−	/$D−	730
*Debug information On	$D+	/$D+	730
*Turbo Pascal map file Off	$T−	/$T−	738
Turbo Pascal map file On	$T+	/$T+	738
*Force far calls Off	$F−	/$F−	731
Force far calls On	$F+	/$F+	731
*Overlays allowed Off	$O−	/$O−	736 V5.0
Overlays allowed On	$O+	/$O+	736 V5.0
*Align data Byte	$A−	/$A−	730 V5.0
Align data Word	$A+	/$A+	730 V5.0
Var string checking Relaxed	$V−	/$V−	739
*Var string checking Strict	$V+	/$V+	739
*Boolean evaluation Short circuit	$B−	/$B−	730
Boolean evaluation Complete	$B+	/$B+	730
*Numeric processing Software	$N−	/$N−	736
Numeric processing Hardware	$N+	/$N+	736
Emulation Off	$E−	/$E−	731 V5.0
*Emulation On	$E+	/$E+	731 V5.0
Link buffer Disk	$L−	/$L−	733 V4.0
*Link buffer Memory	$L+	/$L+	733 V4.0
Local symbols Off	$L−	/$L−	733 V5.0
*Local symbols On	$L+	/$L+	733 V5.0
Conditional defines (no default)	$DEFINE	/D	770
Memory sizes *(16384,0,655360)	$M	/$M	414,734

Map file specifies the level of detail written into Turbo Pascal map (.MAP) files. .MAP files are human-readable summaries of program address and line number information generated during compilation. By default, map file generation is disabled. The **Map** file item allows you to turn on map file generation and specify how much information the map file is to contain.

Selecting **Map** file brings up a menu of four choices:

- *Off* means that map files are disabled.
- *Segments* specifies that *only* segment name, address, and length information is to be included in the map file.

- *Publics* specifies that the map file is to include the segment information mentioned above, plus all public symbol names in the program (including symbols **USE**d in resident units) and the address of the program's entry point.
- *Detailed* specifies that the map file is to contain all the information mentioned above, plus an additional table providing a 32-bit code address for each line number of source code that was available to the compiler during compilation. As you might imagine, this portion of the .MAP file can be enormous, so unless you need the information you'd best select some lesser degree of detail.

Link buffer specifies whether the Turbo Pascal link buffer is to be kept on disk or in memory. When the link buffer is in memory, link operations proceed more quickly; on the other hand, the link buffer occupies memory that might also be given to the program under development. When the link buffer is kept on disk, link operations go more slowly because disk I/O is a great deal slower than memory access. The choice is yours: The default condition is to leave the link buffer in memory, and this will work well for all but the very largest programs. Once your program grows to the point that you are having a hard time compiling it within the Environment, you should change the Link buffer item to **D**isk.

28.2: THE ENVIRONMENT SUBMENU

Beneath the **C**ompiler submenu on the **O**ptions menu is the **E**nvironment submenu. As you might imagine, these options exist to help you set up the Turbo Pascal Environment to suit your own preferences and needs. For that reason, there are no equivalent source code directives or TPC command-line directives. The things you're configuring with the Environment submenu apply *only* to the Environment. Note that there are minor differences between the 4.0 and 5.0 **E**nvironment submenus. These will be described below.

Backing Up Source Files

Turbo Pascal 3.0 and its predecessors always created a backup file with the .BAK extension every time you saved your source code file out to disk. With Turbo Pascal 4.0 and later, you have the option to turn this feature off. The **B**ackup source files item exists to let you make that choice.

Backup source files (simply called **B**ackup files in 5.0) is a toggle, and can be set to either **O**n or **O**ff. The default is **O**n, which means that a backup file will be created each time you save the editor file out to disk. The file as it existed before the edit began becomes the .BAK file, and the new file as written to disk takes on the old file's name.

Automatic Saving Before Running a Program or "Shelling Out"

For those who can't always remember to save early and often, Turbo Pascal provides a safety feature in the form of automatic edit file saving before leaving the editor to do dangerous things.

By "dangerous things," I mean running the compiled program or "shelling out" to DOS using the **O**S shell item on the **F**iles menu. In both of these cases, the Turbo Pascal Environment loses all control over what happens to the machine, and you, the programmer, regain the opportunity to lock the system up by running a faulty program or doing hazardous things from the DOS command line.

The **E**dit auto save item on the Environment submenu enables or disables this feature. The default condition is **Off**, which means you have sole responsibility for saving your source code files out to disk. Pressing E changes the state of the item, and if **On** will save your edit file before running the compiled program or before shelling out to DOS.

Automatic Saving of Compiler/Environment Configuration Files

If you don't explicitly save options configured through the **S**ave options item on the **O**ptions menu, you lose them when you leave Turbo Pascal. The **C**onfig auto save item allows you to have the Environment save your options to disk automatically when you exit. The default condition of this item is **Off**, meaning that the options are *not* saved. Turning it **On** will save all options to a disk file whenever you exit Turbo Pascal to DOS.

The default name of this disk file is TURBO.TP, but you may name it anything you like when you save it to disk. See the discussion of the **S**ave options item in Section 28.4.

Retaining the Saved DOS Screen (4.0 only!)

Ordinarily, when you bring up the Turbo Pascal Environment, the DOS text screen that exists when you invoke the Environment is saved in a memory buffer so that it can be restored when you shell out to DOS or exit the Environment completely. This does require a certain amount of memory for the buffer; 4000 bytes for a 25-line screen; 6880 bytes for a 43-line EGA screen; or 8000 bytes for a 50-line VGA screen. Configuring the number of screen lines is explained below, in connection with the **S**creen size item on the **E**nvironment submenu. The default condition is to retain this screen through your entire session.

If you're pushing the limits of memory and would rather the buffer memory be released for the use of the compiler, you can set the **R**etain saved screen item to **Off**.

Default is **On**. With **Retain saved screen Off**, the Environment will initially save the screen in a buffer, but will release the buffer to serve other needs as soon as memory gets tight.

Note that this item does *not* appear in the 5.0 Environment submenu.

Changing the Size of Editor Tabs

As described in Section 26.3, the Turbo Pascal editor supports either "smart tabs" whose spacing varies depending on the contents of the file, or "hard tabs" which are the traditional tab stop every n characters, where n defaults to 8. The number of characters between tab stops may be set to any value from 2 to 16 by using the **T**ab size item.

When you select Tab size, a dialog box named Editor Tab Size appears, displaying the current tab size value. You can type in any value between 2 and 16, which will become the new tab size for "hard tabs."

Choosing Zoomed Windows or Split Screen

The Turbo Pascal Environment supports two different windows: One for the editor and one for the output from programs. These can both be partially displayed at once using a split-screen technique, or they can be displayed in an "either-or" manner using a feature called *window zooming.*

The **Z**oom windows item controls this duality. When **Z**oom windows is **Off** (the default), both windows are displayed on the split screen. When **Z**oom windows is **On**, only one window is displayed at a time.

The F5 key is a shortcut equivalent to this menu item.

Changing the Screen Size

Most early PC-compatible display adapters only display 25 lines of 80 characters. IBM's EGA, introduced in 1984, can display as many as 43 lines at once, while the IBM VGA and its compatibles, introduced in 1987, can display as many as 50 lines.

The Turbo Pascal Environment defaults to a 25-line screen. However, using the Screen size item you can change the number of lines on your screen to a larger number *if your installed display adapter supports it.* Turbo Pascal can detect the presence of an EGA or VGA display, and will allow you to set the larger screen sizes only for those adapters that support the larger sizes.

Under 4.0, if you select Screen size, a new submenu will appear with three items on it: **25**-line standard display; **43**-line EGA display, and **50** line VGA display. If you have an older CGA or MDA display, only the top item will be accessible. The other two will be visible, but will be displayed in a fainter shade to indicate that they are "dormant."

Similarly, if you have an EGA, only the 25 and 43 line items will be available. The VGA allows you to choose one from any of the three.

The 5.0 menu is similar, except that the 43- and 50-line items are combined into a single item reading

```
43/50 line display
```

The change here is that you can have a 43-line display under the EGA and a 50-line display under the VGA, but the VGA will no longer (as it does under 4.0) allow you a 43-line screen.

When you choose a screen size, the screen changes instantly to that size, and will remain at that size until you change it again or leave the Environment. To keep a screen size across sessions, you must save out your options to a disk file using the **S**ave options item on the **O**ptions menu.

28.3: THE DIRECTORIES SUBMENU

In a system like DOS that supports subdirectories, knowing where things are is critical. Turbo Pascal offers you considerable flexibility as to where you store the compiler file, your unit files, include files, and so on. Managing this flexibility is the job of the **D**irectories submenu of the **O**ptions menu.

When selected, the **D**irectories item brings up a new submenu, which is a little different from other submenus in the Environment, in that it has space to the right of the menu items for the text that they represent (see Figure 28.2).

This menu allows you to specify pathnames of five different directories for the use of the Environment. It also displays and allows you to change the name and path of the current Pick file (see Section 25.3).

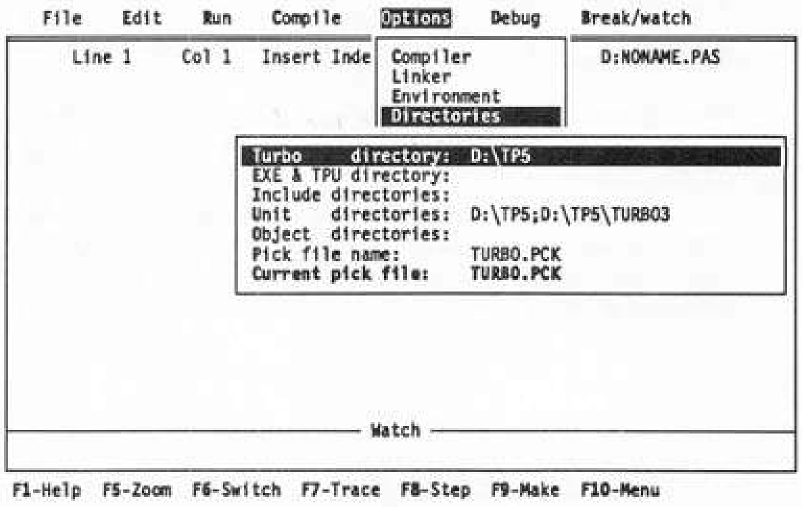
Specifying the Directory for the Turbo Pascal Product Files

It's a good idea to set up a subdirectory to contain the TURBO.EXE file, the TPC.EXE file, the TURBO.HLP help file, and so on. This is the *home directory* for Turbo Pascal. Obviously, if you make the home directory the Turbo directory, the Environment will have no trouble finding any of its various parts and support files.

However, if you set up a separate subdirectory for each of your major programming projects, there must be some way to tell Turbo Pascal how to find its own files like the configuration file, the help file, and so on. You can put the home directory on your DOS directory path list, but that will only allow you to invoke TURBO.EXE or TPC.EXE from anywhere in your system. The files needed by the Environment will not be affected by the DOS path.

Figure 28.2

The Directories Submenu



```
File   Edit   Run   Compile   Options   Debug   Break/watch
  Line 1    Col 1   Insert Inde  Compiler            D:NONAME.PAS
                                 Linker
                                 Environment
                                 Directories
                   Turbo      directory:  D:\TP5
                   EXE & TPU directory:
                   Include directories:
                   Unit     directories:  D:\TP5;D:\TP5\TURBO3
                   Object  directories:
                   Pick file name:        TURBO.PCK
                   Current pick file:     TURBO.PCK

                              Watch

F1-Help  F5-Zoom  F6-Switch  F7-Trace  F8-Step  F9-Make  F10-Menu
```

The Turbo directory item on the Directories submenu allows you to specify this home directory. When you select the item, a dialog box named Turbo Pascal Directory will appear, and allow you to enter a directory path to serve as the home directory. When you press Return after entering the path, the dialog box will disappear, and the path you entered will be displayed to the right of the Turbo directory item in the menu.

You can only enter one home directory path.

Specifying the Directories for Compiled .EXE Files

Ordinarily (and I think wisely), the .EXE and .TPU files generated by the compiler are written into the current directory. You can change that directory to another directory if you choose by the Executable directory item. (Under 5.0, this item is called "EXE & PPU directory.") Again, selecting Executable directory brings up a dialog box that allows you to enter a directory path. Note that, unlike the other directory options defined below, you can enter only *one* directory for executable files, since it is a "write to" rather than a "read from" option.

Specifying the Directories for Include Files

The compiler will always search the current directory for an include file specified by the **$I** compiler directive. You have the option of specifying additional directories to be searched for include files that are not found in the current directory. This may be done through the **I**nclude directories item.

Selecting **I**nclude directories brings up a dialog box into which you type the names of all directories you want the compiler to search for include files. You have 128 characters in which to specify all directories, which isn't all that much if you plan to specify lots of directories or only a few that are nested *way* down deep inside the DOS directory tree.

Multiple directories are separated by a semicolon *only. Do not leave any spaces between one directory and the next.* If you do, the space will cause the compiler to ignore any subsequent directory information.

For example, if you type the following two directory paths into the dialog box, only the first will be recognized by the compiler:

```
C:\TURBO4; C:\JIVETALK
```

Specifying the Directories for Unit Files

Just as with include files, the compiler will always search the current directory for any .TPU files named in a **USES** statement. You may wish to keep generally useful units in a specific place or places, however, rather than copy them all into every project directory you have. In fact, doing otherwise is woefully wasteful of your disk space.

Note that you don't have to specify a directory in order to use units that have been moved into TURBO.TPL system library file. This typically includes **DOS**, **Crt**, **Printer**, **Turbo3**, **Graph3**, and the special **System** unit, but not the BGI unit **Graph**.

But for other units (and this includes **Graph**) you might want to specify a location. This might be the home directory (I put all generally useful units there) or in a special development subdirectory for that unit, where its source code also resides.

The **U**nit directories item allows you to enter one or more directories separated by semicolons into its dialog box, to a maximum of 128 characters. *Do not separate the directories by spaces, or anything but a single semicolon.* You will see no error if you do, but any directories named after the space will be ignored and will not be searched for units named in your **USES** statements.

Specifying the Directories for .OBJ Files

Turbo Pascal links object code generated by 8086 assemblers into its own code by the use of the **$L** directive and the **EXTERNAL** reserved word (see Section 24.6). When the compiler encounters a **$L** directive, it first searches the current directory for the .OBJ file named in the directive. If the named .OBJ file doesn't exist in the current directory, the compiler will search any directory paths you have specified using the **O**bject directories item on the **D**irectories submenu.

As with the other directory items described above, you can specify as many directories as you like, to a total length of 128 characters. Separate directory paths with a single semicolon, and, as I have emphasized above, *do not put any space or other characters between the named directories.*

Specifying a Pick File

The Environment keeps a "history" of the last several files to be loaded into and saved out from the editor, in the form of the pick file (see Section 25.3). The default name of the file is TURBO.PCK, and it is written into the current directory any time you save a file from the editor to disk.

You can change the name of the current pick file to something other than TURBO.PCK if you choose. The means to do this is the **P**ick file name item. As with the other items in the **D**irectories submenu, selecting **P**ick file brings up a dialog box into which you type the name of your new pick file.

Why would you want more than one pick file name? If you are working on a project that involves a main program file, five or six unit files, and a handful of assembly language files, using the pick file is a good way to choose one of the files without having to search for it on a Load file display. Having one configuration file for each large project allows you to have a separate pick file for each large project.

I will discuss the practical issues surrounding large projects in Chapter 29, Project Management.

28.4: OTHER OPTIONS MENU OPTIONS

Specifying Command-Line Parameters

When you run a newly compiled program from within the Environment, there is no obvious way to simulate the command-line parameters that you would normally pass to the program from the DOS command line. The **P**arameters item on the **O**ptions menu allows you to set up a set of parameters for testing purposes while you are working within the Environment.

Selecting **P**arameters brings up a dialog box entitled Command Line Parameters. You may enter up to 128 characters of command line text (which is all that DOS allows) and press Return. The next time you select **P**arameters, the text that you entered will be displayed in the dialog box. You may append to it or edit it as you desire. The parameter text is saved along with the TURBO.TP configuration file (or whatever name you give it) so you can have a set of "standard" command-line parameters for each project you're working on.

Note that you must *not* include the name of the program being run as part of the parameter text.

Loading and Saving Configuration Files

Once you have spent some time and thought configuring the Environment optimally for a given project, it makes sense to save it to disk for reuse later on. The **S**ave options item on the **O**ptions menu allows you to do this.

Selecting **S**ave options brings up a dialog box containing the name of the current configuration file. The default name is TURBO.TP, and Turbo Pascal will open and load a configuration file named TURBO.TP from the current directory each time you load Turbo Pascal. Therefore, you should save any "global" configuration (that is, common to all your projects) options in a file named TURBO.TP. If you have an EGA display you might, for example, keep a common screen size of 43 lines at all times. This kind of information should be kept in TURBO.TP.

Other configuration data, like a set of command line parameters, pick file name, and so on, are specific to a given project and should be kept in a project-specific configuration file. The best thing to name your project-specific configuration file is the name of your project's main source file with a .TP extension.

Alternatively, if you do as I do and confine each major project to its own subdirectory, you can name the project-specific subdirectory TURBO.TP, and the project's configuration data will be loaded automatically whenever you invoke Turbo Pascal from within the project directory.

If you already have a configuration file by a certain name and try to save it by the same name, the Environment will ask for confirmation before overwriting the previous version of the file.

Loading a configuration file is done with the **L**oad options item on the **O**ptions menu. If you work on a lot of projects within a single directory (not an especially good idea unless they are related to one another in some important way), you will have to use **L**oad options to bring in a project-specific configuration file for one of the several projects within the directory.

29 Project Management

I have said in many places that a computer language is a tool for managing complexity. Hiding the details of a process behind a procedure or function name allow you to think of that process as an integral whole without being distracted by internal details that implement the process. By choosing what details to hide and what details to reveal at what level in a program design, you can read that program at any level without being overwhelmed by details belonging to the next level down.

There is a second kind of complexity in computer programming, and that is the logistical complexity of creating a very large software project that cannot be compiled as a single monolithic source code file. As soon as you must divide a program into parts, the number of individual files to be tended explodes, since for every part of the program there will be a source code file, an object code file, a .BAK backup file, and (optionally) a .TPM file.

Keeping track of these files is what Turbo Pascal's project management features were created to do.

29.1: INTEGRAL MAKE AND BUILD

Splitting off portions of a program into separately compiled units is best done along functional lines. For example, a massive accounting program might have four major functions: accounts payable, accounts receivable, payroll, and budget. Putting each of the four major functions in a unit makes for a small, easily understood main program and manageable modules for the four functions. Another advantage to breaking down the program this way is that the four major function units may be specified as (Version 5.0) overlays (see Section 17.5) to reduce the memory requirements of the program as a whole.

Another reason for separate compilation is to move libraries of generally useful procedures and functions into their own units, where they may be accessed in a compiled state and not recompiled each time the program as a whole is compiled.

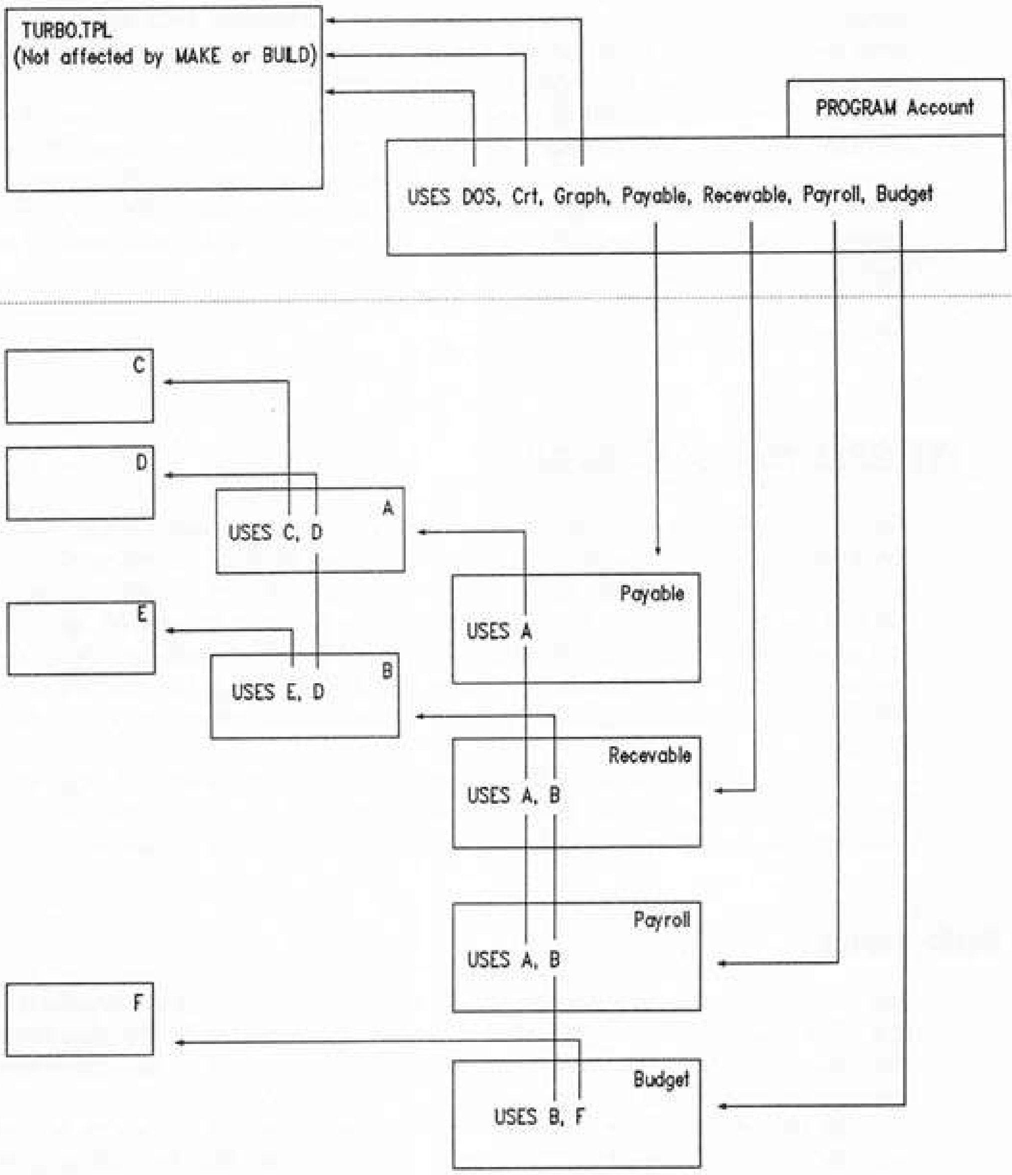
The Build Feature

The accounting program example is shown in Figure 29.1. Six utility libraries named **A**, **B**, **C**, **D**, **E**, and **F** are variously shared by the four functional units, **Payable**, **Receivable**, **Payroll**, and **Budget**. The arrows indicate visually which units use what other units. These relationships between units are called *dependencies*.

By the way the Turbo Pascal compiler works, the USE-er cannot be compiled until the USE-ee is compiled. This implies a definite order that the various units must be compiled if you choose to compile them all at one time. This order is reflected in Figure 29.1 by the units' position relative to the top of the figure. Unit **C** is the first to be compiled because unit **A** depends on unit **C**, and unit **Payable** depends on unit **A**. When compiling, the compiler threads its way down through this *dependency chain*

Figure 29.1

Compile Order in a Build Operation



until it reaches bottom, and then it climbs back out, compiling as it goes. Only after all the units are compiled is main program **Account** compiled.

This operation, of compiling all *compilands* (a compiland is any compilable source file, either unit or main program) contained in the program is called a *build.* You can initiate a build operation by selecting the **Build** item on the **Compile** menu.

For a build to be completed without errors, *all* source code files that contribute anything to the program must be available to the compiler. This includes include files added to a program through the **$I** compiler directive, and .OBJ files linked into a program through the **$L** directive.

The only exception to this is for units that reside in the system library, TURBO.TPL. Units stored in TURBO.TPL may be **USE**d by a compiland without themselves having to be recompiled during the build.

The Make Feature

A build is very much what Turbo Pascal 3.0 did every time you pressed C: It compiled every line of source code in the entire program, regardless of what parts of the program had been changed since the last compile. Compiling everything every time is wasteful and unnecessary, and can be avoided by using Turbo Pascal's integral Make feature.

A *make* is a selective build. The Turbo Pascal compiler inspects the files in the dependency chain, and recompiles only those parts of the program that need compiling.

What does the compiler check? Primarily DOS *time stamps,* that is, the fields in every file's directory entry that indicate when the file was last modified. If a compiland's source code file was modifed more recently than its object code file, that means that you've made changes to the source code file since the last time the file was compiled. The object code file does not reflect those changes, and therefore must be recompiled.

If, on the other hand, the object code file is *newer* than the source code file, it means no changes have been made to the source code file since the last time it was compiled, and the object code file is therefore up to date and in no need of recompilation.

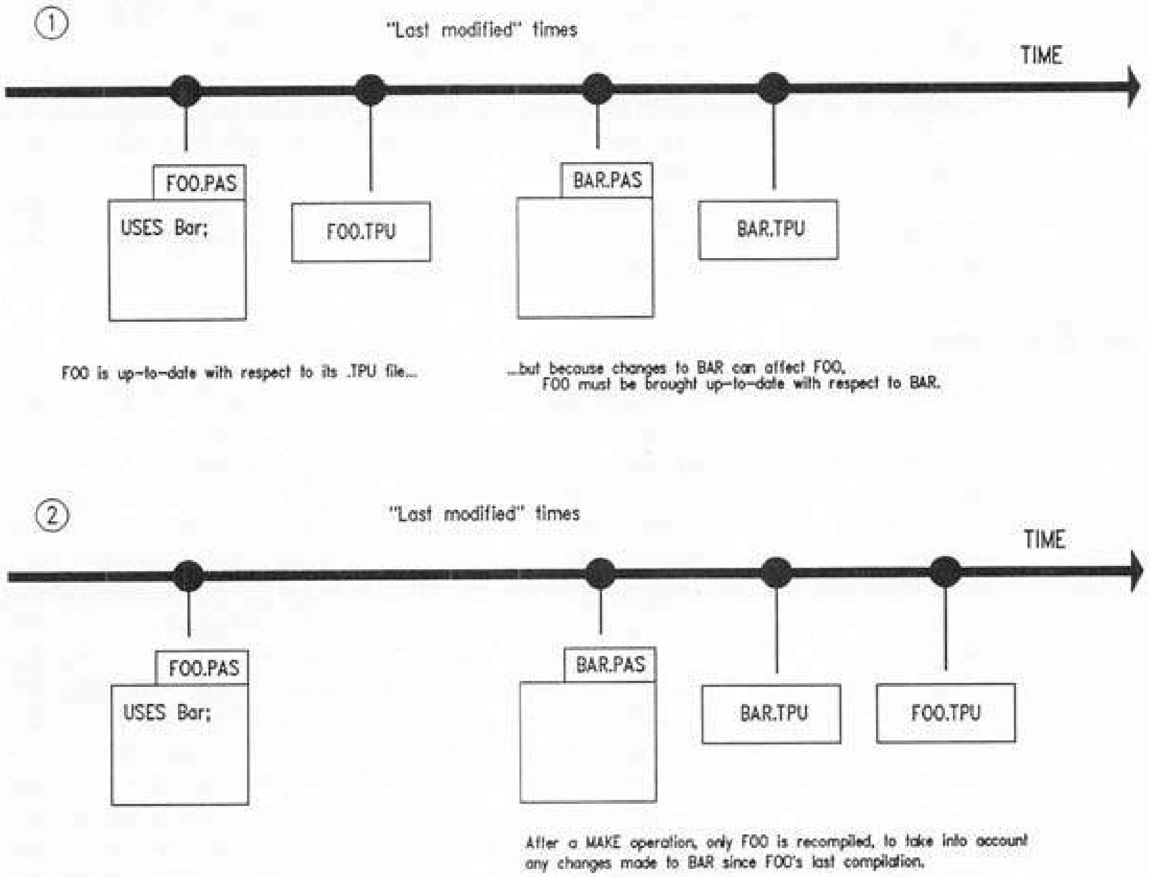
The compiler makes this check individually on every unit file down the dependency chain from the primary file. It recompiles any file whose .TPU file is older than its source file. Note that in *any* make operation, the primary file is *always* compiled, whether or not its object code file is up to date.

Making sure every object code file is up to date with respect to its source code file is necessary, but it isn't enough. The second check the compiler makes as it moves along the dependency chain is that the interface part of any unit is older than the object code file of anything that uses it. This is much more subtle and will take some additional explanation.

Figure 29.2 will help. We have two units along a time line, with the dots on the time line marking the points at which each file was last modified. "Later" is to the right. Both units are individually up to date, in that FOO.TPU is later than FOO.PAS, and BAR.TPU is later than BAR.PAS. *However,* unit **Foo** uses unit **Bar**. Changes may have been made to BAR.PAS after the last time that FOO.PAS was compiled. These changes could invalidate the interface between the two units.

Figure 29.2

Inter-Unit Dependencies and Make


1
"Last modified" times
TIME
FOO.PAS
USES Bar;
FOO.TPU
BAR.PAS
BAR.TPU
FOO is up-to-date with respect to its .TPU file...
...but because changes to BAR can affect FOO, FOO must be brought up-to-date with respect to BAR.
2
"Last modified" times
TIME
FOO.PAS
USES Bar;
BAR.PAS
BAR.TPU
FOO.TPU
After a MAKE operation, only FOO is recompiled, to take into account any changes made to BAR since FOO's last compilation.

For example, unit **Bar** could contain a procedure **Crunchit**, originally defined this way:

```
PROCEDURE Crunchit(InVal,OutVal : Integer);
```

Unit **Foo** would have been compiled to use procedure **Crunchit** as defined above. Then, BAR.PAS could have been modified after **Foo** was last compiled, to add an additional parameter:

```
PROCEDURE Crunchit(InVal,OutVal,ErrVal : Integer);
```

Now, the interface to procedure **Crunchit** is no longer what **Foo** expects it to be. To be sure that all changes to **Bar** are "known" to unit **Foo**, FOO.PAS must be recompiled so that **FOO.TPU** is newer than **BAR.TPU**.

If Turbo Pascal's Make feature detects the situation at bullet 1 of Figure 29.2, it will recompile FOO.PAS so that FOO.TPU is newer than BAR.TPU, as is the case at bullet 2. This way, all changes made to BAR.PAS will be recognized and synchronized with FOO.PAS.

Make does this all the way down the dependency chain. It makes sure that any unit using another unit is "newer" than the unit it uses. Returning to Figure 29.1 for a moment, a make operation would ensure that PAYABLE.TPU was newer than A.TPU, and that A.TPU was newer than both C.TPU and D.TPU. Furthermore, PAYROLL.TPU must be newer than A.TPU, B.TPU, C.TPU, D.TPU, and E.TPU, because all of those units are farther down the dependency chain than PAYROLL.TPU.

What this means is that changing the interface part of C.TPU would cause the Make feature to trigger compilation of every unit shown in the figure except for F.TPU, because F.TPU in no way depends on C.TPU.

Now, why do I specify the interface part? A unit communicates with other units and the main program only through its interface part. Anything defined in the implementation part of a unit is private to that unit and invisible to the "outside world." So changing the implementation part of a unit does not affect the way that unit links up with the other units and the main program. This isn't to say that changing a unit's implementation part has no effect on a program's operation; it only means that the smart linker will not be affected by changes in any unit's implementation part.

The Make feature performs a third and final check on every file in the dependency chain. If a unit file loads a .OBJ assembly language file through the **$L** directive, or if it loads an include file with the **$I** directive, the make operation ensures that the unit is recompiled if either the include files or the .OBJ files are newer than the unit's TPU file. This guarantees that any changes made to the .OBJ files or the include files are recognized by and reflected in the unit's .TPU file.

The nicest thing about the integral Make feature is that it's entirely automatic. You don't have to tell it what to do. Make knows how to inspect time stamps and compare them to find out which is newer. Make also knows which files depend on which other files, and knowing all that, it can compile whatever needs to be compiled to ensure that the program as a whole is up to date without compiling anything unnecessarily.

Creating Dependency (.DEP) Files (4.0 Only!)

Under Turbo Pascal 4.0, you can inspect the dependencies existing in your own programs by generating a .DEP file with the TPMAP.EXE utility program included with Turbo Pascal:

```
TPMAP <filename> /D
```

Here, **<filename>** is the name of a .TPM file generated by the Turbo Pascal compiler. To generate a .TPM file you must have the **$T+** option in force when you compile a program or unit.

The .DEP file produced will show all dependencies, including those involving units residing in TURBO.TPL. Remember, however, that units residing in TURBO.TPL are *not* affected by Make or Build.

This feature has been dropped from Turbo Pascal 5.0.

29.2: CONDITIONAL COMPILATION

Also new to Turbo Pascal 4.0 is *conditional compilation*. By using conditional compilation directives, you can tell the compiler to compile certain portions of a source code file and ignore others, based on symbols that you define.

An example is the best way to approach conditional compilation. I have a relatively simple ASCII text file lister utility called **JList2**. It prints a file neatly under page headers listing the file name, size, and last modified date and time. I designed **Jlist2** for use with laser printers, and it makes use of the "line printer" fonts many of the latest crop of laser printers support.

Now, the problem with using any printer-specific feature (and this by no means is limited to laser printers) is that you have to build printer-specific code or data into the lister program. Most printers respond to escape sequences (that is, series of characters initiated by the ESC character, $27) to change fonts, line spacing, and so on. As you might expect, each printer responds to a different set of escape sequences to do the same thing. There are simply no standards.

Ordinarily, you'd either have to have a separate source code file for each printer you want to support, or else juggle sets of incompatible escape sequences inside the program. There's another option that I've found attractive, and that is to use Turbo Pascal's conditional compilation to use a single source code file to generate several printer-specific versions of the resulting .EXE file.

Consider this: The HP Laserjet II laser printer is reset to its default state using the escape sequence ESC E. The Canon LBP-8 A1 laser printer is reset using the escape sequence ESC c. Supporting both printers can be done easily by using conditional compilation:

```
{ This version compiles if "HPLJII" is defined }
{$IFDEF HPLJII}
PROCEDURE ResetPrinter;

BEGIN
  Write(LST,Chr(27)+'E');
END;

{$ENDIF}
```

```
{ This version compiles if "CANONA1" is defined }
{$IFDEF CANONA1}
PROCEDURE ResetPrinter;

BEGIN
  Writeln(LST,Chr(27)+'c')
END;

{$ENDIF}
```

Here are two different versions of the same procedure. Obviously, both can't be compiled at once, or you would receive an error for duplicate identifiers on **ResetPrinter**. However, you can direct Turbo Pascal to compile only one of the two by defining only one of the two symbols **HPLJII** and **CANONA1**. If the symbol **HPLJII** is defined, the first of the two procedures (everything between **{$IFDEF HPLJII}** and **{$ENDIF}**) will be compiled, and the second will be ignored. If **CANONA1** is defined, the second procedure (everything between **{$IFDEF CANONA1}** and **{$ENDIF}**) will be compiled and the first will be ignored.

Symbols

Now, what *is* a *symbol*, and how are they defined? A symbol is just that: A name that means something to the compiler. It is *not* a part of the program and takes no room in your data segment. If you name the symbol to the compiler, the compiler considers it defined. Otherwise, the symbol is unknown to the compiler and considered undefined.

When working within the Environment, you can define symbols from the **Conditional** defines item on the **Compiler** submenu of the **Options** menu. All you need do is enter the symbol's name in the dialog box that appears. More than one symbol can be defined by separating the symbols with semicolons. However, leave no intervening spaces between symbols.

When using TPC and working from the command line, use the **/D** directive to define symbols:

```
C:\TURBO>TPC JLIST2 /DCANONA1
```

If you later wanted to compile the same source code file to support the HP Laserjet II, you would invoke TPC this way:

```
C:\TURBO>TPC JLIST2 /DHPLJII
```

Again, leave no space between the **/D** directive and the symbol name. The **/D** directive can take multiple symbols, separated by semicolons, or you can pass multiple **/D** directives to TPC with one symbol defined per directive. The choice is yours.

There are a number of predefined symbols that are available whether or not you define them, as summarized in Table 29.1:

Table 29.1
Predefined Symbols

Symbol	*When Defined*
VER40	Always defined if running V4.0
VER50	Always defined if running V5.0
MSDOS	Always defined for DOS versions
CPU86	Always defined for Intel '86 processors
CPU87	Defined if an 87-family math coprocessor is detected at compile time.

The version symbols allow you to test which version of the compiler is in command. This can be useful if you're distributing source code to people who may be using Turbo Pascal 4.0 or Turbo Pascal 5.0 (or, in the future, some later version). You might want to take advantage of certain V5.0-specific features like overlaid units if your user has V5.0. Otherwise, you need to keep the source code syntax to that recognized by V4.0.

As new versions of Turbo Pascal appear, new symbols like **VER51**, **VER55**, or **VER60** will be defined.

The **MSDOS** symbol is always defined for DOS versions of the compiler. This is the only operating system supported for the time being, but if a future version of Turbo Pascal appears for OS/2 or Unix, a symbol like **OS2** or **UNIX** would probably be defined in each version to indicate to the compiler which operating system is hosting the compile. This feature might allow you to cross-compile the same source code file for different operating systems.

Similarly, the **CPU86** symbol is always defined for Intel 86-family CPUs, and currently this is the only version available. However, I would imagine that in future versions of Borland's Turbo Pascal for the Macintosh, a symbol would be defined as **CPU68** to indicate that the native machine was a Motorola 680X0 device.

The **CPU87** symbol is considerably more useful, especially if you distribute source code to be compiled by many users on different kinds of machines. When the compiler begins compiling your code, it checks to see if an Intel math coprocessor is installed. If the compiler finds a coprocessor, it defines **CPU87**; if not, the symbol remains undefined. This allows a single source code file to use different means to optimize floating-point code depending on the presence or absense of a math coprocessor.

Conditional Compilation Based on Symbols

As mentioned above, a symbol is either defined or undefined. There is a separate test for a symbol's being defined and for its being undefined. The **$IFDEF <symbol>** directive causes the code which follows it to be compiled if **<symbol>** is defined, down to the **$IFDEF**'s matching **$ENDIF** directive. The **$IFNDEF <symbol>** directive, by contrast, causes compilation of the code that follows it is **<symbol>** is *not* defined.

For example, suppose you want to build diagnostic display routines into a program for use during testing stages only. During the test, the statements need to be added to a program; later on they must be removed. You could, for example, display the value of a variable this way:

```
{$IFDEF TESTMODE}

Writeln(LST,'<Diagnostic:> MedianAge =',MedianAge);

{$ENDIF}
```

If you compile with **TESTMODE** defined, this statement will be inserted into the executable program file. Later on, compile without defining **TESTMODE** and the statement will no longer be compiled. (This sort of thing is much better handled by the integrated debugger in Turbo Pascal 5.0).

For either/or situations there is an intermediate **$ELSE** directive that separates two alternate segments of source code, one to be compiled if the other is not. For example, if you want to use slightly different code for positioning the cursor with a mouse than with the keypad, you might do something like this:

```
{$IFDEF MOUSE}

IF ButtonPressed THEN
  BEGIN
    GetMouseXY(X,Y);
    GotoXY(X,Y)
  END;

{$ELSE}

IF InChar IN CursorKeySet THEN
  BEGIN
    DeltaXY(InChar,X,Y);
    GotoXY(X,Y)
  END;

{$ENDIF}
```

Here, you either use the mouse or you don't, and each case has its own particular means of processing cursor position information.

In the example at the beginning of this section, I used a separate **$IFDEF** directive for each of two different printers, rather than the **$ELSE** directive. The reason was that I might want to add support for a third or a fourth printer eventually, and with **$IFDEF/$ELSE** you can only choose one of *two* different alternatives. If I wanted to choose one of several different printer reset strings I could set it up this way:

```
{$IFDEF CANONA1}
  ResetString := Chr(27)+'c';
{$ENDIF}
{$IFDEF HPLJII}
  ResetString := Chr(27)+'E';
{$ENDIF}
{$IFDEF LJ400}
  ResetString := Chr(27)+'@';
{$ENDIF}
{$IFDEF PTIGER}
  ResetString := Chr(27)+'*R';
{$ENDIF}
```

If you wanted, you can add additional conditional clauses later on to support additional printers. One caution here is that *you* are responsible for ensuring that only *one* of the various printer symbols is defined at a time. In the earlier example, the compiler would stop on a duplicate identifier error. However, if in this example more than one symbol is defined, the compiler will compile the statement corresponding to each defined printer, and only the *last* of them will be the one that actually carries its value into the rest of the program. *No error will be generated.*

Defining and Undefining Symbols within the Source Code File

You have the option of defining and undefining symbols within your source code itself using the **$DEFINE** and **$UNDEF** directives. The following directive:

```
{$DEFINE TESTMODE}
```

is completely equivalent to defining symbol **TESTMODE** from the command line using TPC, or from the Options menu using the Environment.

Similarly, you can undefine a symbol this way:

```
{$UNDEF TESTMODE}
```

If **TESTMODE** is not already defined, the **$UNDEF** directive does nothing.

Conditional Compilation Based on Compiler Options

Turbo Pascal's conditional compilation feature also has the ability to test the state of switch-type compiler options, and will either compile or ignored blocks of code depending on the state of a particular option. The **$IFOPT <option>** directive does this. **$IFOPT** is the function equivalent of **$IFDEF** except that it tests a compiler option rather than a symbol.

The best example involves floating-point support under Turbo Pascal 4.0, which does *not* have the math coprocessor emulation feature of V5.0. If you want to write a program that works efficiently on machines with and without math coprocessors, you need to use conditional compilation.

The problem centers around the incompatible types **Real** and **Double**. **Real** is the 6-byte software-only real, which is the only floating point type available on machines without an Intel math coprocessor. **Double** is the double precision hardware real number type, which requires a math coprocessor in order to be used in a program. The **$N+** option enables the use of hardware reals, and the **$N−** option disables the use of hardware reals.

In defining groups of numeric variables to be used in floating point calculations, you can use conditional compilation based on the state of the **$N** option to determine whether to define the variables as type **Real** or type **Double**:

```
VAR
{$IFOPT N+}
  R,S,Perimeter,Area : Double;
{$ELSE}
  R,S,Perimeter,Area : Real;
{$ENDIF}
```

$IFOPT can test the state of the **$B**, **$D**, **$F**, **$I**, **$L**, **$N**, **$R**, **$S**, **$T**, or **$V** options.

30

Debugging with Turbo Pascal 5.0

The single most important thing Turbo Pascal 5.0 adds to its predecessor is an integrated high-level debugger. Debugging is something that all Pascal programmers have to do, but until now, there have been few tools to help them do it. Most of us continue to resort to sticking **Writeln** statements here and there in the code to display the value of a maverick variable as the program code unfolds and does its work. This is definitely the hard way to do it. Let me show you the easy way, and you'll never go back.

30.1: THE DEBUGGING LEXICON

The debugging process brings to Turbo Pascal a whole new suite of terms and concepts. People with experience in C or assembly language will be familiar with most of these terms, but people coming from BASIC or who have never stepped beyond Pascal may find the new lexicon confusing in the extreme. Before getting into the details of how to use Turbo Pascal's new debugging features, it will help to define some terms and describe the debugging process from a height.

Low-Level Versus High-Level Debugging

Debugging is nothing more than seeing what your program code and variables are doing as a program runs. If you can watch a bug happen, you can fix it. The problem is that a program, when running, is very much a black box. All you see are its output; words or graphics written to the screen or printer. The program's innards cannot be examined unless to take deliberate steps to examine them.

So a debugger's job is simply to let you look inside a program while it runs. This looking can take place from two different perspectives:

FROM PASCAL'S PERSPECTIVE

A Pascal program consists of Pascal statements and Pascal variables. The vast majority of program bugs can be detected simply by watching the sequence in which statements execute along with the values as they change within program variables. Debugging that limits itself to the examination of language statements and variables is called *high-level debugging.*

FROM THE MACHINE'S PERSPECTIVE

A computer program of any stripe is a series of machine-code instructions that act on locations in memory. The CPU chip executes the machine instructions, which act upon internal memory bins called *registers* as well as external memory bins that we call *system memory.* A portion of system memory set aside for short-term storage is called the *stack.* Each statement in a Turbo Pascal program is equivalent to some number of machine code instructions, and each Turbo Pascal variable is some number of bytes of memory somewhere in the system memory map. *Low-level debugging* involves examining the CPU registers, the stack, the individual machine code instructions, and the individual bytes that make up system memory as the program runs.

In a sense, high-level debugging is looking down on your program from above, while low-level debugging is looking up at your program from beneath. It's the same program in either case, but from above it appears as a sequence of Pascal statements, and from beneath as a sequence of machine instructions.

As you might imagine, low-level debugging is somewhat more difficult but *much* more potent. Borland International sells a low-level debugger called Turbo Debugger that works very well with Turbo Pascal 5.0. I don't have the space here to describe low-level debugging, which merits a book all to itself. The rest of this chapter will limit itself to the high-level debugging features built into Turbo Pascal 5.0.

Breakpoints

People who cut their teeth on interpreted BASIC probably did a lot of debugging by control-breaking out of program execution, and then using the **RESUME** statement to continue execution after examining or changing some variables. Something like this is possible in Turbo Pascal (though much more powerful and tidy) using the concept of *breakpoints*. A breakpoint is a flag raised at some program statement, indicating the program is to halt temporarily at that statement. When the program halts, the Environment takes control, putting all its resources at your disposal. You can examine and change variables and do other things before restarting the program on its way once more.

Stepping Versus Tracing

Turbo Pascal also allows you follow the step-by-step execution of a program by placing a highlight bar over the line of source code that is currently being executed. You can then execute program statements one at a time by pressing a function key. Each time you press the function key, the highlight bar moves to the next Pascal statement in sequence.

You have two ways to execute any statement. *Stepping* through the statement treats the statement as a single step, and executes it. If the statement is a subprogram call, the subprogram is executed normally, that is, without treating *its* statements as steps to be executed individually while you watch. Essentially, the subprogram is executed as one "lump" and you do not get to see the flow of individual instructions within the subprogram.

Tracing a statement, on the other hand, enters subprograms and allows you to single-step a subprogram's statements as well, with a pause between each while you decide what to do next.

While debugging a program, you can step over a subroutine call or trace into it as you desire, simply by choosing the proper function key.

Watches

A *watch* is a window into a program variable, showing you the variable's value and how it changes during program execution. Each time you execute a single program statement, the value in the watch window for a given variable will be updated, if the executed statement changes that variable. If you have a watch open on a counter variable, you can watch its value grow as the program increments the counter variable. If that counter variable doesn't seem to be counting, you can see what's happening to it and find out what's going wrong.

Perhaps one statement increments the counter normally, but another statement later on unexpectedly *decrements* the variable. With a watch on the variable, you'll be able to see the variable first increment up to the next higher value, then decrement down to its original value. Furthermore, you'll be able to tell exactly which statement is erroneously decrementing the counter, because when the counter decrements, the highlight bar will be resting on the statement that does the decrement.

Evaluation

It may not be necessary to continuously monitor the value that changes within a variable, as happens with a watch. You can at any point *evaluate* a variable or an expression by bringing up an evaluation window. The evaluation window will allow you to enter the name of a variable. It will immediately display the current contents of that variable. Furthermore, you have the opportunity of *changing* the contents of that variable as a means of testing the effects of the change on program operation from that point on.

In general, watches are used while stepping or tracing, and evaluation is used when you run and halt a program by means of breakpoints.

Display Swapping

A problem arises when you try to watch a program's innards: Where do you watch them? The program, presumably, is using the screen to communicate with its user. Using the screen to examine statement execution and variable contents will disrupt the screen's normal output.

Turbo Pascal does something called *display swapping* that effectively shares a single physical screen between the Environment, where debugging takes place, and the program's output screen. Each time the program writes to its screen, the Environment saves the updated display out to a buffer, so that the display can be restored when you want to see it, or when the program updates it again.

30.2: WATCHING A SIMPLE PROGRAM RUN

With all of those terms defined and fresh in your mind, it makes sense to load up a simple program and trace through its execution, just to get an initial feel for the high-level debugger. A good sample program is ROOTER2.PAS, which was first presented in Section 13.1. It's a very simpleminded program that requests an integer value from the keyboard and then displays the square root of that value. It repeats this action until you enter a value of zero, which signals the code that it's time to halt and return to DOS.

```
PROGRAM BetterRooter;

VAR
  R,S : Real;

BEGIN
  Writeln('>>Better square root calculator<<');
  Writeln;
  R:=1;
  WHILE R<>0 DO
    BEGIN
      Write('>>Enter the number (0 to exit): ');
      Readln(R);
      IF R<>0 THEN
      BEGIN
        S := Sqrt(R);
        Writeln('  The square root of ',R:7:7,' is ',S:7:7,'.');
        Writeln
      END
    END;
  Writeln('>>Square root calculator signing off...')
END.
```

Load **BetterRooter** into the editor as you would if you were simply intending to compile and run it. Now bring down the **Run** menu: Alt-R is a good way to do it. Note that having a **Run** menu at all is new to Turbo Pascal 5.0. The **Run** keyword in the menu bar of Turbo Pascal 4.0 simply ran the program without offering any additional choices in the matter.

Before trying to debug the program, compile and run it (if you haven't before) just so that you know what to expect from it. That done, let's trace through it one statement at a time.

Begin the trace by selecting the Step over item on the **Run** menu. (F8 is a shortcut for Step over, by the way, and is also the way you continue stepping once stepping has begun.) The **Run** menu will disappear and a highlight bar will appear over the **BEGIN** word at the start of the main program block. Stepping and tracing *always* begin at the start of the main program block. Constant, type, and variable definition statements do not take part in the actual step-through of the program.

A single step is taken by pressing the F8 function key once. *When you press F8, the highlighted statement is executed.* I put that sentence in italics because you *must* keep that straight: the highlight is placed on a line *before* the line is executed, and the line is *not* executed *until* you press F8 (or F7, if you are tracing rather than stepping).

Something that may seem a little odd will happen when you execute most statements during a step-through. The Environment will vanish for a moment, and the output screen (essentially the screen you left behind when you loaded Turbo Pascal from DOS) will appear very briefly. Just as briefly, the output screen will vanish again, and the Environment will return, with the highlight bar on the next statement in sequence.

What gives? This is Turbo Pascal's display swapping in action. Because many statements have the power to write to the display, the Environment, before it executes a statement, will "bring in" the output screen from its buffer and put it into display memory. This is *not* done so that you can see it (so don't complain that it comes and goes too quickly!) but rather to allow the executed statement to write to the display. As soon as the executed statement has done its thing, the Environment saves the output screen back to its buffer and reappears on your display.

On an 8088-based PC, the swapping happens slowly enough for you to glimpse what's happening on the output screen. On AT-type machines (and especially 386-based machines) display swapping will happen so quickly that the screen simply flashes, and that's all you'll see of the output screen. Remember, the output screen is just "passing through" for the program's benefit, and you are not expected to be able to see it.

If you want to see the output screen, all you need to do is press Alt-F5. The output screen will appear instantly. Pressing Alt-F5 (or any other key) will bring back the Environment.

You may also notice that display swapping does not happen on *all* statements. The Turbo Pascal Environment does its best to avoid swapping the display unless necessary, so on some statements that cannot possible write to the screen (e.g., a simple assignment statement) the statement will execute without the Environment swapping the display. This is called "smart" display swapping, and is the default condition. Later on we'll explain how display swapping may be disabled altogether, or done on every single statement.

So, step away! Every time you press F8, the highlight bar will move down by one line. There is a loop in **BetterRooter**, and you can watch execution go around the loop again and again, calculating the square root for a different number each time.

You'll notice that when you execute the **Readln** statement, the output screen comes in and stays in. While **Readln** waits for your input, the output screen will remain in view. As soon as you enter some value and press Return, the Environment will return.

Step through to the end of **BetterRooter** by entering a 0 at the **Readln** statement. When you step off the end of the program, the highlight bar will disappear.

30.3: TRACING A PROGRAM WITH SUBPROGRAMS

BetterRooter is an extremely simple program. It has only one loop and no procedures or functions. Far more of the power of the high-level debugger comes into play when you trace a program that has subprograms.

Let's move ahead by debugging a fairly simple program presented in Section 16.13, **MorseTest**. This program demonstrates Turbo Pascal's sound routines by translating text strings into Morse code.

```
PROGRAM MorseTest;

USES Crt;

TYPE
  String80  = String[80];

{$I SENDMORS.SRC}

BEGIN
  ClrScr;
  SendMorse('CQCQCQ DE KI6RA *SK',850,15);
END.
```

The main program of **MorseTest** contains only two statements. The first clears the screen, and the second actually sounds the Morse code equivalent of the string parameter through the speaker. Load the program and step through it as we did with **BetterRooter**, using F8 to execute each of the two statements in turn. When the first statement executes, the output screen will clear. When the second statement executes, you will hear Morse code from the system speaker. After executing the call to **SendMorse**, the step-through is finished and the highlight bar will disappear.

You've probably already decided that this won't help you much if you have a problem inside the **SendMorse** procedure. Fortunately, there is a way to step through subprograms as well as main program blocks: Trace into (the shortcut is F7).

Begin another step-through of **MorseTest**, but this time, press F7 to execute each statement. The **ClrScr** statement will execute just as before. However, when you execute the call to **SendMorse** by pressing F7, the **MorseTest** program vanishes from the screen and the source code for the **SendMorse** procedure appears in its place, with the highlight bar over the opening **BEGIN** of the **SendMorse** procedure body.

The Difference between Trace into and Step over

This is a good place to ponder the difference between **Step over** (F8) and **Trace into** (F7). Step over executes the highlighted statement *as a single entity*. In other words, a procedure call is executed as a procedure call without any attempt to step through the procedure's statements as well. **Trace into**, by contrast, follows subprogram calls down into the subprograms, and will single-step all program statements (including subprogram statements) as it encounters them.

You are not obliged to use one or the other during a step-through of a program. Each time you see that the highlight bar is over a subprogram call, you can decide which way to go: To step over the subprogram call (executing it without single-stepping it) or to descend into the subprogram and single-step its statements as well.

So give it a try. When the **SendMorse** procedure call is highlighted, press F7 rather than F8. The source code for **SendMorse** will be loaded into the editor, and you can begin single stepping the procedure.

The bulk of procedure **SendMorse** is one enormous **CASE** statement. You might expect that the **CASE** statement would be executed in one blow (because only one of its case labels can be found equal to the case selector), but Turbo Pascal bends its rules a little bit and acts as though each individual test within the **CASE** statement were itself a separate statement.

So once you enter the **CASE** statement, each press of F7 moves one case label down into the **CASE** statement until the case selector matches the case label. Only then does the highlight bar skip to the end of the **CASE** statement and highlight the next statement in sequence.

The statement that immediately follows the **CASE** statement is another procedure call, to a local procedure named **Morse**. When the call to **Morse** is highlighted, you can take your choice: To step through **Morse** or simply execute it. If you choose to execute it, you will hear a generated Morse code character sound off on the speaker. Choosing to step through **Morse**, however, will demonstrate something important: Single-stepping will disrupt programs (like **MorseTest**) that depends on system timing.

Try it and see. Procedure **Morse** uses Turbo Pascal's sound-generating procedures and the **Delay** procedure to create audible Morse code characters. The **Sound** procedure turns on the tone through the speaker. The **Delay** procedure is supposed to dictate how long the tone is to remain on. Once you execute the **Sound** procedure, however, the tone will remain on while the Environment waits for you to press F7 again. Stepping through **Morse** will create some sound, but it won't necessarily sound like Morse code. Keep this effect in mind: Single-stepping a program isn't always possible without seriously altering the program's intended output. Fortunately, your ability to execute a procedure call without single-stepping the procedure allows you to avoid stepping through those parts of the program that won't take single-stepping gracefully.

Statements Versus Source Code Lines

Up until now I haven't drawn any distinction between a program statement and a line of source code, but the distinction must be made. The Turbo Pascal integrated debugger is a *line-oriented* debugger. It executes one *line* of source code with each single step rather than one *statement.*

I have on occasion combined several statements onto one source code line, particularly assignment statements that assign initial values to a whole clutch of program variables. Cramming many statements onto one line is generally bad practice from a readability standpoint, and now, with integrated debugging, you have even more reason to avoid overdense lines of code. One statement per line is the most readable format you can write, and also the most amenable to sanity in debugging with Turbo Pascal 5.0.

Tracing into Units

The **MorseTest** program in the single-stepping example above brings in an include file containing its single subprogram. The include file has to be present for the program as a whole to compile, so the compiler can with confidence assume that the include file can be loaded in for single-stepping.

But what about subprograms existing in units? The compiler will try to trace a subprogram in a unit by searching for the source code for that unit. If the source code for the unit is found, it will be loaded into the editor and tracing will continue. If the source code is *not* found, no error is generated, but the subprogram will be executed without tracing through it, exactly as it would be if **Step over** were executed rather than Trace into. Note that units residing in SYSTEM.TPL are *never* traced, whether or not their source code is available to the the compiler.

30.4: EXAMINING VARIABLES THROUGH EVALUATION AND WATCHES

Single stepping allows you to see the order in which statements are executed, which can often be a great help in deciding what it is wrong with a balky program. Statements are only half of what a program is about, though. Looking "inside" a variable is just as important as single-stepping statements, and Turbo Pascal provides plenty of power in this area as well.

Using the Evaluation Window

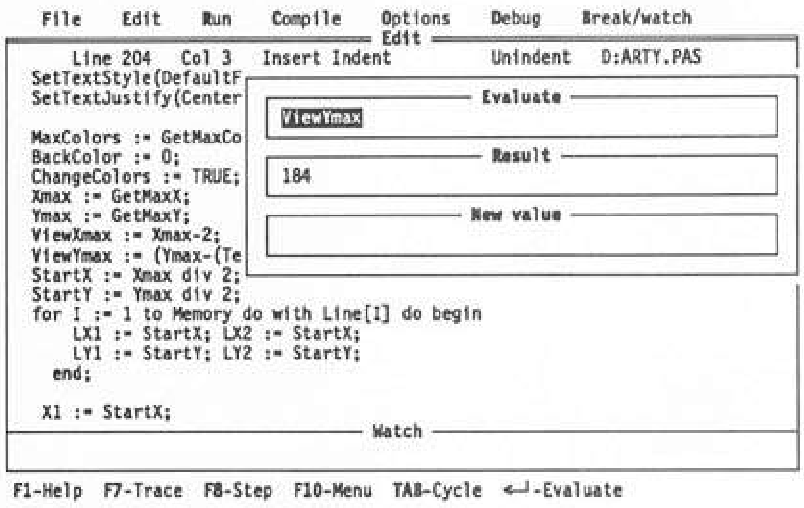
While the Environment is paused, waiting for you to execute the next statement via F7 or F8, you have the ability to examine any program variable in or above the current scope. This is done by bringing up a window especially for that purpose. The window can be invoked from the **Debug** menu, as the top item, **Evaluate**. (Ctrl-F4 is the shortcut for **Evaluate**.) The evaluation window is divided into three fields, as shown in Figure 30.1. Each field of the evaluation window has a name, indicating its function.

The top field of the evaluation window is the *Evaluate field*. It accepts the identifier or the expression that you wish to evaluate. The middle field is the Result field, which displays the result of evaluating what you enter into the top field. The bottom field is the New value field, into which you can enter a new value for a modifiable identifier currently shown in the Evaluate field.

When you bring up the evaluation window, perhaps at a breakpoint during a debug session, the word containing the editor cursor will be in the Evaluate field. This does not guarantee that the identifier is an evaluatable item; it may be the reserved word **BEGIN**, for example, or the name of a subprogram, neither of which make any sense in the context of the Evaluate field. If the identifier in the Evaluate field is *not* what

Figure 30.1

The Evaluation Window



```
  File    Edit    Run    Compile    Options    Debug    Break/watch
                                Edit
      Line 204   Col 3   Insert Indent        Unindent   D:ARTY.PAS
   SetTextStyle(DefaultF
   SetTextJustify(Center                     Evaluate
                          ViewYmax

   MaxColors := GetMaxCo
   BackColor := 0;                           Result
   ChangeColors := TRUE;  184
   Xmax := GetMaxX;
   Ymax := GetMaxY;                          New value
   ViewXmax := Xmax-2;
   ViewYmax := (Ymax-(Te
   StartX := Xmax div 2;
   StartY := Ymax div 2;
   for I := 1 to Memory do with Line[I] do begin
       LX1 := StartX; LX2 := StartX;
       LY1 := StartY; LY2 := StartY;
     end;

    X1 := StartX;
                                Watch

 F1-Help  F7-Trace  F8-Step  F10-Menu  TAB-Cycle  ←┘-Evaluate
```

you want to evaluate, you can move the cursor around the screen, and place it at the start of any evaluatable identifier. Then, when you bring up the evaluation window, the identifier that starts under the cursor will be in the Evaluate field. You need only press Return to evaluate it.

A further extension to this slick trick is that you can copy additional text from the edit window to the Evaluate field by pressing the right arrow key. When you want to evaluate an entire expression, this will be a valuable feature, because the Evaluate field will initially only copy in the single word containing the cursor.

Of course, you also have the option of typing in the name of an identifier to evaluate. If you wish to evaluate an expression that does appear anywhere on the screen, typing it in will be your only option.

The most obvious use for the evaluation window is in inspecting and modifying program variables. Any time your program is paused, either through a breakpoint or between single steps, you can enter the name of an identifier in the Evaluate field and its value will be displayed in the Result field. This allows you to peek at your program variables whenever you want.

One of the most desirable features of the evaluation window is its ability to display the values of data items that have no easily printable form. A set, for example, will be displayed as the values present in the set between the set builder brackets. For example, a set of characters containing the upper and lower case 'Y' characters plus the BEL character would be displayed as:

```
[#7,'Y','y']
```

If the displayed value is wider than the 50-odd characters that can be seen in a field at one time, the field may be scrolled rightward and leftward as desired, using the arrow keys.

The evaluation window will also evaluate expressions, within certain limits. The expression must be a true expression, in that it must "cook down" to some single value. In addition to program variables it can involve numeric, character, and Boolean literals; constants; any arithmetic or logical operator; and certain built-in functions. The list of valid functions is not long, and it is not obvious which built-in functions are valid and which are not. The list of functions that I have found to be valid include: **@**, **Abs**, **Addr**, **Chr**, **CSeg**, **DSeg**, **Hi**, **Length**, **Lo**, **Odd**, **Ofs**, **Ord**, **Pred**, **Ptr**, **Seg**, **SPtr**, **SSeg**, **Succ**, and **Swap**.

Even this short length of functions allows you to do some very useful things, like locating program identifiers (including subprograms) in memory; checking the length of strings; and watching the stack pointe register. Unfortunately, all of the functions that act on real numbers are excluded from expressions. This includes **Pi**, the trig functions, and the integer/real transfer functions.

The evaluation window can also act as a primitive programmer's calculator, in allowing you to determine the result of masking operations, shifting quantities, and so on, simply by typing the expressions to be evaluated; for example,

```
$F4 SHR 3

133 OR 51

MyMask AND $CB
```

The window will convert hexadecimal values to decimal; all you need do is type a hex value like $D3 into the Evaluate field, and its decimal equivalent will appear in the Result field. Sadly, you can't force it to work the other way, and convert decimal to hex.

The evaluation window will reject an Evaluate field that cannot be evaluated with an error message in the Result field, typically, "Field cannot be evaluated." Also, it will not allow you to modify a nonvariable in the New value field.

Display Formats and Format Directives

Turbo Pascal provides considerable flexibility in how you display the data contained by the variables you evaluate. The default display format for each type of data is the one that makes the most sense, but there are *format directives* that you can use when evaluating that will display data in a slightly different way. These format directives are characters that are appended to the data item in question with a comma, as in:

```
BelChar,c
```

The format directives are summarized in the table below:

Directive	*Function*
c	Displays "control characters" 0–31 as their resective IBM-standard symbols rather than the default of a pound sign followed by the character's numeric equivalent, i.e., character 1 will be shown as a "smiley face" rather than as #1. Works *only* on characters or elements of strings.
d	Dumps the preceding data item (which may be of any type) as a series of hexadecimal values.
fn	**n** specifies the number of visible digits for display of any floating point types.
p	Overrides the default pointer format **PTR($<seg>,$<ofs>)** to a simple (and shorter) display of **<seg>,<ofs>** with both in hex. May be used with any address type or expression that cooks down to an address, such as **Addr,** and @. Legal but unnecessary for **CSeg, DSeg, Seg,** and **Ofs.**
r	Applies the record field label to the value of each field in the display of records, as in **LASTNAME : 'Duntemann'** rather than merely **'Duntemann'.**
x	Forces integer types to display as hexadecimal rather than the default of decimal. Legal but ignored for other simple types; i.e., you cannot force control characters to be displayed in hex with **x;** use **d** instead.

Characters are displayed in one of two ways: *printable* characters are shown in their standard ASCII form, in quotes, as in 'B'. *Control* characters 0 through 31 are displayed as their numeric equivalents with a preceding pound sign, as in #7 for the BEL character 7. You can override this format using the c format directive, which forces the use of IBM standard symbols for control character display. Characters in the high-order character set are displayed in quotes using the standard IBM symbols, as in '╔' for character 201.

Sets are displayed as you would build them, within set-builder brackets, with the elements of the set separated by commas. If the compiler identifies a closed interval within the set, the items in the closed interval will be displayed as a closed interval rather than as individual elements. Given our enumerated type **Spectrum** from Section 8.2, a **SET OF Spectrum** might be displayed as shown below:

```
[RED..YELLOW,INDIGO]
```

Note that even though we originally defined the constants of type **Spectrum** in mixed case, they will always be shown in upper case when displayed in the Evaluate field, because the compiler forces all program identifiers to upper case before it processes them. This is true for the constants comprising an enumerated type in all circumstances, not simply in set display.

Booleans are displayed as one of the two upper case constants **TRUE** or **FALSE**.

Pointers are displayed using the syntax of the built-in **Ptr** function:

```
PTR($6180,$30)
```

You can use a terser display format by appending the **p** format directive to a pointer identifier, as in **MyPtr,p**. This form simple displays the hexadecimal formats of the segment and offset values, separated by commas:

```
6180,30
```

Note that you can dereference pointers in the Evaluate field by using the familiar pointer notation **MyPtr^**, which will display whatever data **MyPtr** points to, rather than the address **MyPtr** contains. Dereferencing can also be done to generic (untyped) pointers, but *only* if you include the format specifier **d**. This will display a hex dump of the 86 bytes following the address to which the untyped pointer points (86 because each byte requires three characters to dump, and 3 × 86 fills the 255 character limit of the Result field).

Records default to a list of field values separated by commas and enclosed within parentheses:

```
(NIL,'Sprightly',42,PTR($A134,$131))
```

You can also display the field names in front of each value by appending the **r** format directive to the end of the record identifier:

```
(Next:NIL,KeyName:'Sprightly',KeyVal:42,Prev:PTR($A134,$131))
```

Records that contain nested records will be displayed with the inner records enclosed in parentheses.

Arrays are displayed, like records, enclosed by parentheses with commas separating individual values. Most arrays are too long to display entirely within the Evaluate field, and the compiler will show only the part of the array that will fill 255 characters. For an integer array, this is only 86 elements, which is not very much! Displaying some portion of the array that does not fall within the first equivalent 255 characters requires a little more finesse. You need to specify the starting element to display, followed after a comma by the number of elements to be displayed. For example, to show ten elements of an array starting at element 1,677, you would enter this into the Evaluate field:

```
MyArray[1677],10
```

If you intend to use a format directive in addition to the repeat count value, the format directive must come *after* the repeat count.

File variables are displayed with the status of the logical file followed by the name of the physical file (if any) connected with the logical file. The status is given as one of the four uppercase words **OPEN, CLOSED, READ**, or **WRITE**. A file currently being read would be displayed as

```
(READ,'STARDATA.BIN')
```

A logical file that is not connected with any physical file is displayed simply as **(CLOSED)**.

Looking more closely at a file variable's innards can be done by casting the file variable onto a record of type **FileRec**. (Type casting in the Evaluate field will be explained more fully below.) The **FileRec** type is defined in the **DOS** unit, and if you intend to perform such a type cast, your program must **USE** the **DOS** unit *even if you make no other use of anything from the DOS unit.* The cast is entered this way:

```
FileRec(StarFile),r
```

This essentially converts **StarFile** to a record, and will be displayed as any other record would be displayed. The **r** format directive helps comprehension of the display a little bit by preceding each of the **FileRec** field values with the field's name. This will allow you to monitor the file handle, the file mode, the record size, and other file internals.

Modifying Program Variables in the New Value Field

The third field in the evaluation window is the New value field. It will allow you to alter the values of program variables during a program pause, and then restart the program to see what happens when the entered variable takes on its new value. This allows you to play some interesting "what if?" games with malfunctioning programs, by letting you enter what you think is the correct value of a wayward variable to see how the program responds.

With a variable or typed constant in the Evaluate field, you can enter any assignment-compatible value into the New value field, and that value will be stored in the variable or typed constant. This value may be an expression, as long as the expression cooks down to a value assignment compatible with the identifier in the Evaluate field. The New value field performs some range checking and assignment compatibility checking on the entered new value, and will display an error message if the new value is out of range or cannot be assigned to the identifier in the Evaluate field.

You can enter new values only for simple types and strings; in other words, you cannot enter a complete array or record, even a short one that could be entered within the limits of the field. You *can* enter a new value for any simple type or string element of an array or field of a record.

You cannot enter new values at all for sets or files.

Keep in mind that by modifying the values of pointers into system memory, or absolute variables defined over some essential part of system memory, you can crash your system without any protection from the Turbo Pascal Environment. Be careful.

Local Symbols

There is a conceptual difference between global variables and variables belonging to subprograms. Global variables exist in a program's data segment and exist throughout the time the program has control of the machine. Identifiers connected with subprograms are local, and are allocated on the stack only during the period that the subprogram has control. When the subprogram terminates and returns control to its caller, the subprogram is *deinstantiated*, and all of its parameters and local variables are removed from the stack.

Ordinarily, when you compile a program with the intent to debug it, all local symbols are compiled into the Environment's symbol tables. Thus, when you trace into a subprogram, you can evaluate a variable or put it into the watch window (see below on watches). However, to save memory when you are not intending to debug a program, you can turn this access to local symbols off. The means is the **$L−** compiler directive, or the **L**ocal symbols item in the **D**ebug menu.

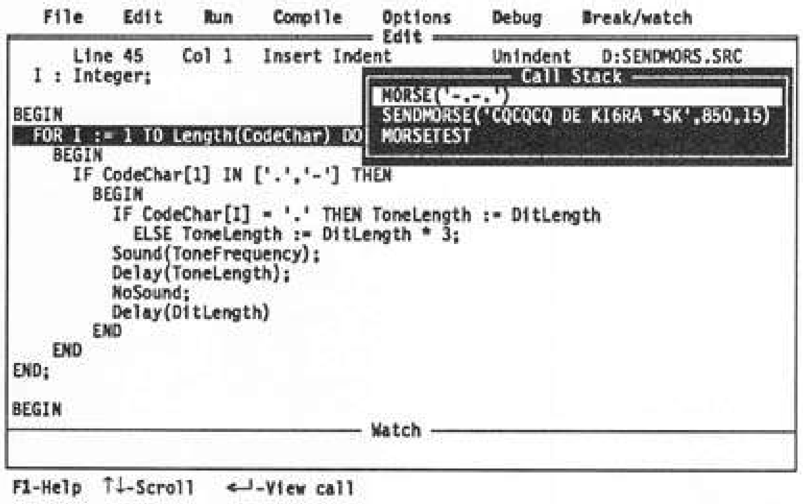
The Call Stack

Another very handy inspection tool offered by Turbo Pascal 5.0 is the *call stack*, which is a small window that will show you how deeply you are nested in subprogram execution, along with (most usefully) the parameters with which the current instantiation of each nested subprogram was called. A screen display showing the call stack from within the **Morse** procedure of the **MorseTest** program is show in Figure 30.2. Note that the call stack builds from the bottom up; the most recently entered subprogram will be on top. Keep in mind that while as many as 128 levels are tracked by the call stack, only nine levels will be displayed at a time. To inspect more than nine, use the arrow keys to move the displayed stack up and down within its window.

Another interesting thing to do is inspect the call stack at various points during the execution of a recursive function, like the **Factorial** function and **QuickSort** procedure from Section 14.4. The names of the subprograms on the call stack will be the same, but the parameter values will be different for each call. The best way to do this is to set a breakpoint (see the next section on breakpoints) at the beginning of the main block of the recursive subprogram, so that you can inspect the call stack during the breakpoint pause each time the recursive routine takes control.

Figure 30.2

The Call Stack



```
   File    Edit    Run    Compile    Options    Debug    Break/watch
                                      Edit
      Line 45    Col 1   Insert Indent          Unindent   D:SENDMORS.SRC
  I : Integer;                        Call Stack
                                    MORSE('-.-.')
BEGIN                               SENDMORSE('CQCQCQ DE KI6RA *SK',850,15)
  FOR I := 1 TO Length(CodeChar) DO MORSETEST
    BEGIN
      IF CodeChar[1] IN ['.','-'] THEN
        BEGIN
          IF CodeChar[I] = '.' THEN ToneLength := DitLength
            ELSE ToneLength := DitLength * 3;
          Sound(ToneFrequency);
          Delay(ToneLength);
          NoSound;
          Delay(DitLength)
        END
    END
END;

BEGIN
                                      Watch

F1-Help  ↑↓-Scroll   ←┘-View call
```

One thing to keep in mind: The call stack will *not* display the names of nested subprograms if you have disabled access to local symbols, by using the **$L–** compiler directive, or the **L**ocal symbols item on the **D**ebug menu. Subprograms are, after all, local entities from the global perspective of the main program. Without access to local symbols, you can't see subprograms any more than subprogram parameters or variables.

Watches

Turbo Pascal offers another window into the private lives of program variables: Watches. A watch is, in essence, a continuous evaluation window into a single variable, updated after each step taken when stepping through a program. Without having to take any ongoing action, you can observe the continuous changes in a variable's value as you step your way through the program.

Watches exist in the watch window, which shares the screen with the Edit window when you debug programs without enabling zooming. A watch consists of the name of a variable, or of an expression that is evaluated and updated after each step taken while stepping a program. Any symbol or expression that may be evaluated in the evaluation window (as described above) may also be a legal watch. The display format and format directives are identical.

Adding a watch to the watch window is done through the **Add** watch item on the **Break**/watch menu. Alternatively, you can use the Ctrl-F7 shortcut to add a watch. When you choose to add a watch by pressing Ctrl-F7, a small window entitled Add Watch will appear. The window will already contain the word under the cursor in the edit screen. Generally, the cursor follows the highlight bar, but you can move the cursor independently of the highlight bar with the arrow keys to select an identifier for the Add Watch window.

You can also type the name of the identifier into the Add Watch window. If you wish to add a format directive to the identifier in the Add Watch window, you will have to type that out explicitly, just as you would in working with the evaluation window, described earlier. Pressing Return will add the watch on that identifier to the watch window.

Once a watch is in the watch window, it will be updated each time you take a program step. Of course, if the value of the watch item doesn't change through the action of a step, it won't change after that step. The Environment evaluates every watch in the watch window after each program step taken, and updates each watch any time its value changes.

You can edit or delete existing watches by switching from the edit window to the watch window by pressing F6. When the watch window is active, a highlight bar will highlight the current watch. This watch can be edited by pressing Return, or deleted by pressing the Delete key. You can also add a new watch to the window by pressing Ins.

While watches are tailor-made for watching variables while single-stepping a program, they operate identically when you run a program with breakpoints. At any breakpoint pause, the Environment will update watches in the watch window before turning control back to you.

30.5: USING BREAKPOINTS

Single-stepping through a program from start to finish (as we've done in the simple examples above) can be enlightening, but it isn't practical for larger projects. The potential tedium of single-stepping through a 10,000-iteration loop should be obvious. What makes a lot more sense is to execute a malfunctioning program normally until execution gets close to the suspected site of the trouble, and *then* begin single-stepping until the source of the problem is found. For this another type of tool is needed: Breakpoints.

A breakpoint is a stop sign inserted into your code by the compiler at your request. When your code is executing and encounters a breakpoint, it will pause and the Environment will take over the screen. The source code will appear in the edit window, with the highlight bar over the source code line containing the breakpoint.

At this point, you can do whatever you need to do; all the resources of the Environment are at your service. You can examine or modify variables using the evaluation window. You can single-step your way further into the program. You can bring the output screen into view (using Alt-F5) to see what the program has accomplished up to that point. Or you can continue full-speed execution of the program until it terminates, or until it encounters another (or re-encounters the same) breakpoint.

Setting and Clearing Breakpoints

Breakpoints are *line-oriented.* In other words, a breakpoint can only be set at the start of a program line, not at some individual statement within a multistatement line. This is yet another reason to confine yourself to one program statement per line.

A breakpoint is set by placing the cursor on the line where you wish to place a breakpoint, and then invoking the **T**oggle breakpoint item on the **B**reak/watch menu. There is a shortcut for **T**oggle breakpoint: Ctrl-F8. Clearing (removing) a breakpoint is done the same way: By placing the cursor on a line that already contains a breakpoint, and then pressing Ctrl-F8. The *toggle* means that the same command will set a breakpoint on a line that doesn't have one, or clear a breakpoint from a line where a breakpoint has already been set. When a line contains a breakpoint, it is set to a different color or highlight level, depending on what sort of screen your copy of Turbo Pascal has been installed for.

Using Breakpoints to Examine Program Operation

There's no better way to get a good feel for how breakpoints are used than just to use them. Let's give it a try, and in the process I'll demonstrate all the various things that can be done using the many facilities Turbo Pascal's debugger can offer.

An interesting program to observe is the **Locate** program described in Section 20.9. Load LOCATE.PAS into the Edit screen, and take a moment to read and become familiar with the general logic of the program, if you haven't already. There is only one note of warning: **Locate** is a relatively advanced program, incorporating a number of DOS calls, recursion, and some pretty involved understanding of the DOS file system. If you're a thoroughly green Turbo Pascal programmer, it might make sense to come back to this section once you've had a chance to bone up on other Turbo Pascal topics and have had some experience in programming larger programs.

Locate is a "whereis" program. It searches your entire hard-disk directory for files that match a given file spec. In other words, if you know you have a little utility program somewhere on your 40MB hard disk called GRIMBLE.EXE, (or was it GRIMBLE.COM?) you can find it by typing:

```
LOCATE GRIMBLE.*
```

Any time **Locate** encounters a file matching the **GRIMBLE.*** filespec anywhere on your hard disk, it will print out the full pathname, size, and time/date stamp of the file to your screen.

The main program of **Locate** is only a frame for a large procedure called **SearchDirectory**. Although **Locate** searches the entire DOS directory tree on each invocation, it only searches one subdirectory at a time, by invoking **SearchDirectory** to search each subdirectory. If, in the course of searching a subdirectory, **SearchDirectory** encounters a child directory, it calls itself recursively to search that child directory.

Before running **Locate**, enter a command-line parameter by selecting the Parameters field of the Options menu. A good one is *.BAK, since almost everyone has some .BAK backup files somewhere on their hard disk system.

Adding Some Breakpoints

Now you're ready to add breakpoints to **Locate**. Before you can do that, however, you need to decide just what you're looking for. In a real debugging situation, an errant program would be doing something odd, or failing to do something that you consider correct. The place to go in the source code has to be your best guess as to the location of the trouble. You may be (or will probably be) wrong on your first guess, but you have to start somewhere. Debugging is not an exact science, but a process of successive refinement of hunches.

Perhaps **Locate** isn't locating all the files it should be locating. The place to go, pretty obviously, is the point at which **Locate** decides that it has found a file to match the search file specification. Once you're there, you can inspect certain variables to see just what's going on.

Locate tests the **DOSError** flag every time it executes **FindFirst** or **FindNext**. If the value is 0, then a file has been found. A good spot for a breakpoint would be immediately after the **FindFirst** invocation that looks for files (an earlier **FindFirst/FindNext** cycle searches separately for subdirectories) in the **BEGIN..END** block executed when **DOSError** is equal to 0:

```
   { Now, make the FIND FIRST call: }
   FindFirst(TempDirectory,0,CurrentDTA);

   IF DOSError = 3 THEN       { Bad path error }
     Writeln('Path not found; check spelling.')

   { If we found something in the current directory matching the filespec, }
   { format it nicely into a single string and display it: }
   ELSE IF (DOSError = 2) OR (DOSError = 18) THEN
     { Null; Directory is empty }
   ELSE
•    BEGIN
       DTAtoDIR(CurrentDIR);       { Convert first find to DIR format.. }
       DisplayData(Directory,CurrentDIR);        { Show it pretty-like }

       IF DOSError <> 18 THEN { More files are out there... }
         REPEAT
           FindNext(CurrentDTA);
           IF DOSError <> 18 THEN  { More entries exist }
•            BEGIN
               DTAtoDIR(CurrentDIR); { Convert further finds to DIR format }
               DisplayData(Directory,CurrentDIR)        { and display 'em }
```

```
          END
      UNTIL (DOSError = 18) OR (DOSError = 2)  { Ain't no more! }
   END
```

The breakpoint position is marked with a square bullet symbol in the first column. You mark a breakpoint by moving the cursor to the desired line and pressing Ctrl-F8, which is a shortcut to the **T**oggle breakpoint item on the **B**reak/watch menu. Note that there are *two* bullets on the code fragment above. Remember, **FindFirst** and **FindNext** operate together. If you want to break on every file found by **Locate**, you must set a breakpoint after each. Breaking after **FindFirst** will happen only on the first file located in each subdirectory. Mark both breakpoints shown.

Adding Some Watches

Next, you need to decide what to examine when you pause for breakpoints. When a file is found by **Locate**, file information is retained in a record called **CurrentDTA**, of type **SearchRec**. How about a watch on **CurrentDTA**? That wouldn't be hard to set up, but for purely practical reasons you would not want to display the whole record, as the first 21 bytes are private to DOS and not especially meaningful. Better would be to set two separate watches on the **Attr** field and the **Name** field, so you can check each file's attribute byte. Adding a watch to the watch window is done by pressing Ctrl-F7, which is a shortcut to the **A**dd watch item on the **B**reak/watch menu. Add those two watches to the watch window.

Executing to a Breakpoint

Now we're ready to do a little breakpoint dancing. Run the program by pressing Ctrl-F9, which is a shortcut to the **R**un item on the **R**un menu. Choosing from the menus gets tedious in a hurry during a debugging cycle. Learn your shortcuts and use them!

What happened? One of two things: Either **Locate** found a file with a .BAK extension, or it didn't. If it didn't (and you are a tidy person indeed!), the highlight bar will not appear at the breakpoint, or be visible anywhere on the screen at all. You'll see the following in the watch window:

```
CurrentDTA.Name: Unknown identifier
CurrentDTA.Attr: Unknown identifier
```

This doesn't mean anything is wrong; it merely means that because no file was ever found, no meaningful values were ever placed in

```
CurrentDTA.
```

More likely, a file will have been found, and the highlight bar will be displayed over the **FindFirst** breakpoint. Down in the watch window, you'll see information on the found file:

```
CurrentDTA.Name: AFILE.BAK
CurrentDTA.Attr: 32
```

The 32 indicates that the file has been changed since it was last copied by an archiving utility. Most of your files will carry an attribute byte value of 32. Files that have *not* been changed since they were last archived will have an attribute byte of 0.

Running from Breakpoint to Breakpoint

So, onward. We could single-step through the program from this point, but that would be tedious and unnecessary. Pressing Ctrl-F9 (the **R**un shortcut) will restart the program. **Locate** will run until it encounters the second breakpoint, and will display another file name and attribute byte in the watch window. By pressing Ctrl-F9 repeatedly, you can restart **Locate** and pause each time a file is found, until no more files are located.

Something to keep in mind, especially if you have some experience with Turbo Pascal 4.0, is that the output window is no longer displayed by default. The watch window has taken its place on the 5.0 display. The output window is still there; however, if you wish to see it you must bring it into view in one of two ways:

- Press Alt-F5. This "swaps" the Environment display for the output window in its full-screen form. Pressing any key then brings the Environment back.
- Switch the active window to the watch window (F6) and then press Alt-F6. The output window then takes the place of the watch window on the Environment display. This is what you can do if you want to observe part of the display while tracing execution in the edit window. Pressing Alt-F6 puts the watch window back into its place beneath the edit window.

Resetting the Program being Debugged

Now let's put **Locate** through another debug run with a different search specification. Select the **P**arameters item from the **O**ptions menu and enter C:\*.COM as the parameter. I want you to specify your boot drive here, as well as *.COM. If you boot from A: instead, use A:\*.COM, etc. We're changing the command-line parameters, but we're probably somewhere in the middle of a program run. **Locate** is paused, but it's still in the middle of executing, with the *.BAK specification in force. To start again from the top with a new search specification, we must reset the program.

This is done using the **P**rogram reset item on the **R**un menu. Selecting **P**rogram reset "wipes the slate", clearing all variables so that the program will begin executing without any assumptions left over from a previous debug run.

Once the program is reset, run it again with Ctrl-F9. **Locate** will show you all the

.COM files on your boot drive . . . or will it? Sharp DOS hackers will notice that the two famous lurkers of the DOS file system, IBMBIO.COM, and IBMDOS.COM, which are always the first two files in any DOS boot directory, will not be displayed. This may be a bug or it may be a feature; which it is depends on what you want **Locate** to do. If you want **Locate** to examine *all* files for the search spec (say, everything matching *.COM) then it's definitely a bug. Let's see how it could be fixed.

IBMBIO.COM and IBMDOS.COM differ from almost all other files in that they have attribute bits 0, 1, and 2 set. These are the Read-only, Hidden, and System bits, respectively (see Figure 20.6 in Section 20). To match these bits, **FindFirst** and **FindNext** must be asked for them. Notice in the code fragment from **Locate** shown earlier, the invocation of **FindFirst**:

```
FindFirst(TempDirectory,0,CurrentDTA);
```

The 0 parameter is the attribute byte we wish to search for. DOS is funny about attribute bits; an attribute byte of 0 will locate files having an attribute byte value of 0 *or* 32; that is, all "normal" files. Setting the attribute byte to 8, to locate the DOS volume label, will find *only* files whose attribute byte is 8. Setting the Read-only, Hidden, and System bits in the attribute byte will find files with those bits and other files as well; *not* setting them will locate only normal files.

Now, change the middle parameter to 7, which sets Read-only, Hidden, and System. Press Ctrl-F9 to begin **Locate** executing. Notice that whenever you change the source code of the program being debugged, Turbo Pascal will recompile the program before running it. If other .COM files exist in subdirectories on your boot volume, they will probably be displayed, but ultimately, you will see the watch window display IBMBIO.COM and IBMDOS.COM. With an attribute byte of 7, **Locate** really will locate *all* files, with the sole exception of volume label files, which (as I explain in Section 20.8) are not really files in the truest sense of the word.

Changing Breakpoints

Finding most bugs won't be quite as easy as *that.* You may discover that your path of inquiry is a dead end, and that stopping at your chosen breakpoints won't help. You can change breakpoints easily, in one of two ways: You can toggle an individual breakpoint off the same way you toggled it on, by placing the cursor on the breakpoint line and pressing Ctrl-F8. If you feel that *all* your breakpoints are completely useless, you can clear them all at once by selecting the **C**lear breakpoints item on the **B**reak/watch menu.

Let's do that, and clear both the breakpoints we've been using thus far. For a new breakpoint, we might decide to watch the point at which **SearchDirectory**, the heart of **Locate**, calls itself recursively. That point is clearly marked in the source code, at the line shown below:

```
{ Here's where we call "ourselves." }
SearchDirectory(NextDirectory, SearchSpec);
```

Set a breakpoint on the **SearchDirectory** line by pressing Ctrl-F8. Also, add a watch to the watch window for the identifier **NextDirectory**. Hint: The shortcut is Ctrl-F7. Remember that we still have the original two watches, on **CurrentDTA.Name** and **CurrentDTA.Attr**.

Set the command-line parameter back to our original *.BAK. Reset the program (this must be done through the **Run** menu; there's no shortcut) and run it (Ctrl-F9). What happens obviously depends on how complex the structure of your hard disk is, but if you have lots of nested subdirectories you're in for an interesting ride.

Remember how **Locate** works: It searches the root directory for subdirectories. If it finds a subdirectory, it calls itself recursively to search the subdirectory. If the subdirectory has a subdirectory, yet another recursive call descends into that subdirectory to search it, and so on (see the detailed discussion of **Locate** in Section 20.9). **Locate** descends into the directory tree until it finds a subdirectory that has no subdirectories. Then it searches that subdirectory for files that match the search specification. That done, it "climbs" back up through the subdirectory tree, searching for more subdirectories, and not searching for files in any given subdirectory until it runs out of subdirectories in that subdirectory.

Each time you press **Run** (Ctrl-F9) **Locate** will execute until it encounters a new subdirectory. Then it will pause, just before it makes a recursive call to itself to search the new subdirectory. Notice the watch window. Once you're a few levels deep, you'll see something like this:

```
NextDirectory:  '\WINDOWS\ACTOR\CLASSES'
CurrentDTA.Name:  'CLASSES'
CurrentDTA.Attr:  16
```

The breakpoint pauses execution just *before* the recursive call is made. When the call is made, **NextDirectory** will contain the path for the search spec passed to **FindFirst**. Note also the attribute value of 16, which indicates a set Subdirectory bit. The directory entry with the name 'CLASSES' is thus a subdirectory, and not an ordinary file or a volume label. Attribute bits are important; it's nice to be able to see them at every step as the program executes.

Tracing into a Recursive Call

Something interesting can happen at this point. Suppose you trace into the recursive call by pressing Trace into (F7). Abruptly, the values in the watch window will change to something like this:

```
NextDirectory: #21'0098GHf'#0'hggJ9'#7#11'H99'#0#0#0'*7^^'#0
CurrentDTA.Name:  'GGHy'#7
CurrentDTA.Attr:  19
```

What gives? All of a sudden, the variables we've been watching have inexplicably turned to garbage. Well, maybe not; think for a moment: by tracing into **SearchDirectory**, we kick off a brand new instantiation of the subprogram on the stack, including all new instantiations of **SearchDirectory**'s local variables. Both **NextDirectory** and **CurrentDTA** are local variables. What you're seeing in the watch window are local variables allocated on the stack but not yet filled with anything, and what the watch window displays is the random trash in memory underlying the newly allocated local variables.

No, the **NextDirectory** value we were watching as a local variable in the previous instantiation of **SearchDirectory** was passed to the new instantiation as a parameter. To verify this, we can bring up the evaluation window, and take a peek at the parameters to this latest instantiation of **SearchDirectory**. Press **E**valuate (Ctrl-F4.) Enter the name of **SearchDirectory**'s first formal parameter, **Directory**. Voila! There's the path value again, in the Result field of the evaluation window:

```
'WINDOWS\ACTOR\CLASSES'
```

Type the second parameter, **SearchSpec**, into the Evaluate field, and there's our faithful, unchanging search specification originally entered as a command-line parameter:

```
'*.BAK'
```

If you continue to single-step through the new instantiation of **SearchDirectory**, you will eventually see the local copies of **NextDirectory** and **CurrentDTA** take on new values, if, of course, any additional child subdirectories are found.

Looking at the Call Stack

How many levels deep are you into the recursive traversal of the directory tree? In our particular example, you can count the number of subdirectories in the path that was just passed to the latest instantiation of **SearchDirectory**. In most cases, however, you won't have such a nice, convenient roadmap telling you where you are in terms of subprogram nesting. Turbo Pascal has such a roadmap, and you can call it up as the **C**all stack item on the **D**ebug menu. (There is no shortcut.) This is the *call stack*, and it will be displayed in a window. The actual data in the window will look something like this:

```
SEARCHDIRECTORY('\WINDOWS\ACTOR\CLASSES','*.BAK');
SEARCHDIRECTORY('\WINDOWS\ACTOR','*.BAK')
SEARCHDIRECTORY('\WINDOWS','*.BAK')
SEARCHDIRECTORY('\','*.BAK')
LOCATE
```

The call stack tells us that we're executing a program called **LOCATE**, and that we're four levels deep in recursive calls to **SearchDirectory.** Furthermore, it shows us the actual parameters passed to each instantiation of **SearchDirectory**.

The call stack can be extremely handy for debugging complex programs that contain many levels of subprogram nesting. Such programs sometimes crash for what seems like no identifiable reason, when a previous (and similar) run of the program operated correctly. What actually happens may have less to do with code bugs than with something else: Stack space.

Monitoring Available Stack Space

A program begins its life with a fixed amount of stack space, either the default 16,384 bytes, or some other amount set with the **$M** compiler directive (see Section 20.10). When you run out of stack space, you crash with a runtime error. It's that simple.

Programs using recursive methods are particularly prone to stack crashes. Each recursive call needs stack space for allocation of parameters and local variables. If you recurse down too many levels, your program will use up its stack allocation and crash. If you write a program that uses recursion, you should keep an eye on your stack space. Here's how, using our current example.

Add a watch to the watch window (the shortcut is Ctrl-F7). This time, the watch will not be a variable or parameter, but the built-in function **Sptr** (see Section 23.1). **Sptr** returns the stack pointer value at any given time. Because the stack pointer starts at the high end of the stack and works its way downward, the value returned by **Sptr** is an accurate indicator of the amount of stack space you have left.

Reset the program and run it again. At the first breakpoint, the watch window will contain a line like this:

```
Sptr: 14476
```

Keep on recursing. The next time **SearchDirectory** calls itself, the value returned by **Sptr** will have changed to 12826. Taking the difference of the two figures will show you that each instantiation of **SearchDirectory** takes *1,650 bytes!* If you start out with 16,384 bytes of stack, you have at best nine levels of recursion available before you crash.

I too was surprised at the 1,650 byte figure for each instantiation of **SearchDirectory**. But consider: **SearchDirectory** contains two parameters of type **String**, and two local variables of type **String**. Each instance of type **String** is 256 bytes in size, and we're up to 1,024 bytes already, just in string storage. Add to that the size of a **SearchRec** (43 bytes), a **DIRRec** (295 bytes), and a **Registers** record (20 bytes) and you're up to 1382 bytes. Then there's the additional overhead of return addresses, pointers to temporary string values, and local storage private to the compiler. Suddenly, 1,650 bytes per call doesn't seem so outlandish.

Quick tip: To determine how large a given variable or data type is, just bring up the evaluation window (Ctrl-F4) and enter an expression like

```
SizeOf(DIRRec)
```

into the Evaluate field. The size of **SizeOf**'s operand will appear in the Result field. Let the computer do what computers do best!

If you start having mysterious stack crashes, put **Sptr** into your watch window and keep an eye on it. If you're like most Pascal programmers, you're not used to figuring out the sizes of variables, nor imagining that stack space is anything but infinite. Like natural resources, stack space is limited. You have to watch it.

Locate is a good program to trace. It does a lot of interesting things, and yet it's simple enough to understand without many hours of study. I'd recommend spending some time with it, setting breakpoints here and there, and using the watch window and the evaluation window to examine its variables at different points of execution. As an exercise, add to **Locate** the ability to locate a subdirectory as well as a file or files. It's not as simple as you might think at first thought, nor as difficult as it might seem after a minute's mulling it over.

An entire book could be written (and probably should) on the art of debugging Turbo Pascal programs. I've done little more than scratched the surface here in hope of getting you interested. Be aware of what the compiler offers. Turbo Pascal's integrated debugger can tell you lots of useful things you never knew how to determine before, and in doing so can keep you out of many different flavors of trouble.

30.6: THE RUN, DEBUG, AND BREAK/WATCH MENUS: A SUMMARY

The **R**un, **D**ebug, and **B**reak/watch menus are specific to Turbo Pascal V5.0. None of them are present in Turbo Pascal 4.0. In this section I'll provide a quick reference to the three new menus. I've described most of the items on the menus during the debugging walk-throughs in the previous section, but it may be handy to have them all in one place.

The Run Menu

The **R**un menu *item,* of course, is present in Turbo Pascal 4.0. But in 4.0 there is no menu behind it; selecting **R**un simply caused your program to run. Turbo Pascal 5.0 presents you with a menu when you select **R**un from the main menu bar at the top of the screen.

The top item on the **R**un menu is ***Run***. Select it, and your program runs. (The shortcut is Ctrl-F9.) If any breakpoints have been set in your program, execution will pause at the first breakpoint encountered.

Beneath **R**un is *Program reset*. Its function echoes its name: It resets the current program and gets everything ready for another run through it, "from the top." If you're stepping through the program and things seem hopelessly muddled, selecting **P**rogram reset is a good way to begin again.

The *Go to cursor* item (shortcut: F4) is a portable breakpoint. Quite simply, selecting **G**o to cursor will execute the program code until execution encounters the statement on the current cursor line. This can be handy if you're stepping through a program and wish to skip ahead by a screen or two. Moving the cursor quickly downward by using the PgDn key and then pressing F4 will execute the code between the current position of the highlight bar and the new position of the cursor.

The Environment will refuse to begin execution if you place the cursor over a part of the program that does not represent executable code, such as constant, type, or variable declarations.

There is one "gotcha" to keep in mind, however: If you move the cursor to a line in the program that isn't in the direct path of execution, execution will keep on going until the program terminates or encounters a breakpoint. For example, if your cursor lands within a **BEGIN..END** block controlled by the **ELSE** clause of an **IF** statement, and the **ELSE** clause is not selected, execution will breeze right by the **BEGIN..END** block containing the cursor.

A corollary of this that should be obvious is that you can't move the cursor *up* in the source file and somehow expect execution to go backwards toward the cursor. If you move the cursor up the source file and press F4, execution will move forward relentlessly until it encounters the end of the program or a breakpoint. Of course, if execution loops *back* to the line containing the cursor, execution will pause, but there is nothing "magical" about **G**o to cursor that alters the essential control path specified by your program.

Trace into (shortcut: F7) single steps to the next line of source code. If the next line is a subprogram, the trace will move into that subprogram. This will happen *only* if the source code for the subprogram is available to the Environment. Also, you cannot trace into a unit compiled with the **{$D−}** or **{$L−}** compiler directives in force, even if the source code is available. These two directives (see Section 27.3) turn off the generation of essential debugging information needed by the Environment to step through the code and examine local symbols.

Also, tracing with **T**race into will not enter external machine code subprograms, interrupt procedures, exit procedures, or **INLINE** macros. Tracing code within an exit procedure may be done by setting a breakpoint at the start of the exit procedure and then beginning to step once normal execution pauses for the breakpoint.

Step over (shortcut: F8) single steps to the next line of source code. If the next line is a subprogram, the trace will skip over the subprogram call. The subprogram call will be made and the subprogram executed normally; however, you simply won't see it occur line by line.

By intelligently choosing either **T**race into or **S**tep over while single-stepping a program, you can examine the subprograms you need to examine and not be bothered tracing the ones that you know work correctly.

The Debug Menu

The **D**ebug menu is entirely new to Turbo Pascal 5.0. It contains most of the selectable items that involve debugging, except for those connected with breakpoints and the watch window.

Evaluate (shortcut: Ctrl-F4) brings up the evaluation window, which is described in detail in Section 30.4. Through the evaluation window you can examine the contents of program constants and variables, either alone or as part of simple expressions. Furthermore, you can modify the contents of typed constants and program variables, so that execution can continue with the new values in place.

Find function allows you to locate the first line of any subprogram without having to scan your source code visually or use the editor search function. This includes subprograms existing in units used by the program, as long as source code for the unit containing the subprogram is available to the Environment. Selecting **F**ind function brings up a window into which you enter the name of the subprogram to be found. Once you enter the name of the subprogram, the cursor will be moved to that subprogram. *The program is not executed.*

You can't use (or even select) **F**ind function unless you have a program compiled into memory. Furthermore, the Environment will not be able to locate a subprogram unless both debugging and local symbols information is turned on (both are on by default).

Note that **F**ind function positions the cursor at the first line of the *body* of the subprogram, not the subprogram header or its constant, type, or variable declarations.

The *Call stack* item was described in detail in Section 30.4, and demonstrated during the trace of **Locate** in Section 30.5. The call stack displays the current nesting level of program execution. In other words, it shows the names of all subprograms into which execution has progressed, including the actual parameters passed to each instantiation of the subprogram. During debugging, the Environment keeps track of up to 128 nested subprograms and their parameters, of which it can display up to nine in the call stack window. (The others may be seen by scrolling the window with the arrow keys.)

The call stack cannot be examined unless a program has been compiled and execution begun.

Integrated debugging is a toggle that specifies whether the compiler generates additional information essential to the operation of Turbo Pascal's integrated debugger. When on, *Integrated debugging* appends information on line numbers and global symbols to the memory image of the compiled program. This information is used by the Environment to single step the code, locate breakpoints, and display global symbols.

Standalone debugging is a toggle that specifies whether or not information will be added to the .EXE file to enable its operation with standalone debuggers like Turbo Debugger, CodeView, or Periscope. This information is primarily concerned with program symbols; that is, the human-readable names for things that exist in your source code but ordinarily do *not* exist in your .EXE files. The default condition is *not* to generate this information, which makes for slightly faster compiles and slightly smaller .EXE files. If you leave Standalone debugging off, you will be able to use a standalone debugger to debug your .EXE file *only* at the assembly-code level, which means that you will be wading through thousands of lines of assembler mnemonics like MOV, POP, AND, etc. Turning Standalone debugging on adds the symbolic information like variable and procedure names to your .EXE file, so that the standalone debugger can display the names of procedures and variables and give you access to items that ordinarily exist only in your source code.

Display swapping specifies the way the DOS output screen is handled while single-stepping a program. Having only one screen is a problem while debugging, since your program (presumably) writes information to the screen while the debugger is tracing the program's execution. Turbo Pascal 5.0 handles this by *display swapping;* that is, by bringing the DOS output screen into view temporarily so that output may be written to it. While the Environment has control of the screen, the DOS output screen is safely stored on the heap. While the DOS output screen is displayed, the Environment's screen is stored on the heap.

There are three modes that govern display swapping:

- *Always.* With this mode in force, the DOS output screen is brought into view *every time* a statement is executed. As soon as the statement is completed, the output screen is swapped back out onto the heap and the Environment screen is swapped back in. This is the mode you should use with programs that write to the display in "sneaky" fashion, as with the **MEM**, **MEMW**, or **MEML** routines. If the DOS output screen is not displayed when these statements execute, the Environment screen will be disrupted, and the DOS output screen will not be characteristic of the program's output.
- *None.* In this mode, no screen swapping is done at all. This mode may be used while debugging programs that do no screen output at all. Such programs are few indeed. If you disable screen swapping using the None mode, any screen output your program performs will disrupt the Environment screen.
- *Smart.* This is the default mode, in which the Environment tries its best to decide which statements will write to the display, and swaps the DOS output screen into view *only* for those statements that affect the display. This sounds terrific, but in practice, Smart mode is conservative and takes few chances. Smart swapping will swap in the output screen on *any* subprogram call, even with subprograms that perform no screen output. This includes most built-in subprograms, again, whether or not they perform output to the screen. What Smart swapping will allow without swapping, by and large, are assignment statements.

Smart swapping is the default. You can change the mode in force by highlighting the **D**isplay swapping item and pressing Return until the correct mode appears beside the item.

Refresh display, when selected, re-displays the Environment screen. In case you have disrupted the Environment screen by debugging a program that writes to the screen while the Never swapping mode is in force, **R**efresh display will repair the Environment's framework and redisplay the current screen of source code in the editor screen.

The Break/watch Menu

This menu, new to Turbo Pascal 5.0, controls the setting and clearing of breakpoints and watches. For more detailed information on breakpoints and watches, see Section 30.4 and Section 30.5.

The first item on the **B**reak/watch menu is *Add watch.* Selecting Add watch through the menu or its shortcut, Ctrl-F7, brings up a window that allows you to enter the name of a program or constant, or an expression, to be added to the watch window at the bottom of the Environment screen.

The ***D**elete* watch item deletes the current watch in the watch window. The current watch is indicated by a bullet (•) in the left margin of the watch window.

The *Edit watch* item allows you to edit the current watch, which is indicated in the watch window by a bullet (•) in the left margin.

The ***R**emove all watches* item deletes all watches currently in the watch window.

The remaining items in the **B**reak/watch menu pertain to breakpoints rather than watches.

The *Toggle breakpoint* item (shortcut: Ctrl-F8) sets a breakpoint on any line where no breakpoint is set; or it will clear a breakpoint when the highlighted line contains a breakpoint. A line containing a breakpoint is displayed at a different color (for color screens) or intensity (for monochrome screens) than lines that do not contain a breakpoint.

For more on setting, clearing, and using breakpoints, see Section 30.5

The *Clear breakpoints* item clears all breakpoints from the file currently displayed in the edit window, in addition to any breakpoints set in units or include files incorporated into that program.

The *Next breakpoint* item moves the cursor to the next breakpoint in line-number sequence in your source code file. This includes breakpoints contained in units and include files. Nothing is executed by this item; the cursor is simply moved to the next breakpoint in the file.

Conclusion

There are a lot more books that could be written about Turbo Pascal. As close as this one has gotten to the size of the Manhattan phone book, it only touches on a multitude of things. DOS interface could support a middling book, and the BGI could easily support one or more. Interrupt-driven programming is a book-sized topic, as are debugging and large project management. Books like these will be turning up from every quarter over the next few years.

But why wait? Whether you can write or not, or whether you ever *will* write or not, the raw materials of books are scattered all over your hard disk and on that pile of paper in the beer box in the corner. You've solved many problems and discovered lots of things about the programming process, and this is the stuff of which books are made. I'm not suggesting that you actually publish a book. What I am suggesting is that you organize your notes, keep good records, and draw diagrams of the stack as you've probed it under peculiar circumstances. When you figure out something that's been hounding you for weeks, *write it down.* Make the effort to explain the solution to yourself, even if your vanity will never allow another pair of eyes to read your prose.

I've been working with Turbo Pascal for nearly five years. Obviously, I know it all off the top of my head, right? *Wrong.* When I get into a new project, I find myself reaching for a book constantly, to remind me how the **BlockRead** procedure works, or to look up the name of the BGI constant for a crosshatch fill pattern. The book I'm currently reaching for is the massive D-ring binder that contains this manuscript. In a few months it will be this book as a real book. Of course I wrote this book primarily for you, the reader. But I'm not exaggerating at all when I say that I wrote it for myself as well.

I'm taking a break from Turbo Pascal books for the time being. My next book will be *86 Assembling from Square One,* to be published by Scott, Foresman and Company in mid-1989, followed by one or more books on object-oriented programming. As always, I welcome your letters (my address is on the order sheet) as long as you'll forgive me for not personally answering every one.

In the meantime, stay organized. Collect your thoughts, and keep them where you can find them when you need them. You will need them. The book that will be of the most value to you is the one you write for yourself. In the meantime, grab a Jolt, crack your knuckles, and get ready to hack some serious Turbo. (It's a magnificent obsession, *ne c'est pas?*)

Index of Programs, Functions, and Procedures

NOTE: The names listed below are not in all cases the same as the names given in the text proper. Instead, longer names have been shortened to follow the convention for MS DOS filenames. These shorter names are the names under which the software appears on the listings diskette. (See front overleaf for ordering information, or check your local user group or CBBS.)
The file extensions are defined as follows:

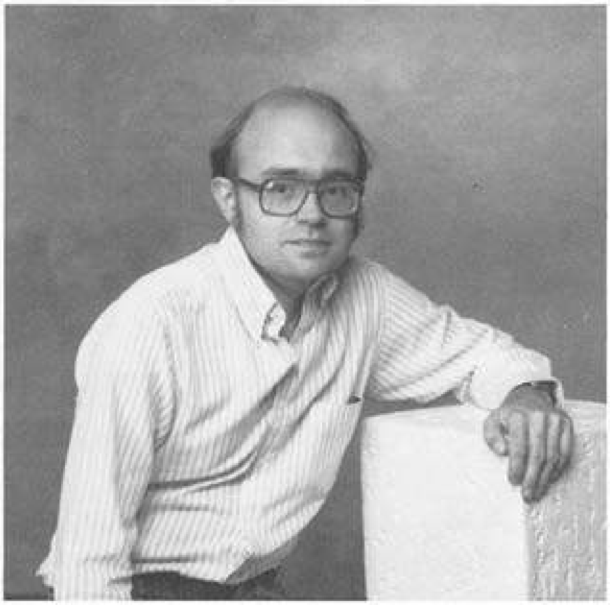
•PAS	A Pascal program
•SRC	A Pascal function or procedure
•ASM	An assembly language routine
•BAT	A DOS batch file
•DEF	A collection of type or constant definitions

Name	*Section*	*Page*	*Name, If Different From Text*
87THERE.SRC	24.1	603	Is87There
AVERAGER.PAS	19.4	317	
BEEP.SRC	16.13	236	
BINARY.PAS	23.5	570	BinaryDemo
BOILER.ASM	24.5	642	
BOXSTUFF.PAS	17.3	250	
BOXTEST.PAS	14.1	177	
BOXTEST2.PAS	17.3	253	
BUTTON.ASM	24.5	662	
CALCDATE.SRC	20.6	377	
CALCTIME.SRC	20.6	375	
CAPSLOCK.SRC	9.6	106	
CASE.PAS	19.7	323	Caser
CLEARS.ASM	24.5	644	
CLREGION.SRC	18.1	277	ClearRegion
CODEMARK.SRC	24.4	637	
CREATLBL.SRC	20.8	399	CreateLabel
CURSOFF.SRC	18.4	294	CursorOff
CURSON.SRC	18.4	296	CursorOn
DATEREC.DEF	20.6	379	
DAYOWEEK.SRC	20.6	384	DateToDayOfWeek
DEMOMENU.DEF	22.9	531	
DIRREC.DEF	20.7	390	
DIRSTRIN.SRC	20.7	392	DIRToString
DISPDIR.SRC	21.4	458	DisposeOfDirectory
DOWNCASE.SRC	9.6	106	
DRAWMARK.SRC	22.3	475	DrawMarker
DTATODIR.SRC	20.7	391	
EVALUTOR.PAS	15.1	206	Evaluator
FACTRIAL.SRC	14.4	186	Factorial

Name	*Section*	*Page*	*Name, If Different From Text*
FASTINX.PAS	24.2	628	FastIncrement
FIBERGLS.PAS	22.4	489	PsychedelicFiberglas
FLIPFLD.SRC	18.3	288	FlipField
FLUSHKEY.SRC	20.3	359	
FLUSHTST.PAS	20.3	360	
FONT.PAS	18.4	297	
FONTS.PAS	22.6	519	
FONTSIZE.SRC	18.4	295	DeterminePoints
FRCECASE.SRC	15.3	215	ForceCase
GAMEBORD.PAS	24.8	671	
GAMER.ASM	24.8	672	
GAMETEST.PAS	24.8	670	
GETDIR.SRC	21.4	450	GetDirectory
GETEXT.SRC	15.1	202	
GETKEY.SRC	20.4	363	
GETLABEL.SRC	20.8	397	
GETSTRIN.SRC	15.2	207	GetString
GFILE.PAS	19.10	336	GraphFiler
GLOAD.SRC	19.10	335	
GSAVE.SRC	19.10	335	
HEXDUMP.PAS	19.10	337	
ISMONO.PAS	13.9	171	
ISWHITE.SRC	9.6	106	
JTERM.PAS	23.7	585	
KEYTEST.PAS	20.4	365	
KSEARCH.SRC	19.9	329	KeySearch
LISTMAN.PAS	21.3	438	
LOCATE.PAS	20.9	406	
LOOK1.PAS	24.2	617	
LOOK2.PAS	24.2	619	
MAIN.PAS	17.4	265	
MORSTEST.PAS	16.13	240	Morsetest
MOUSE.PAS	17.3	255	
MOVLOCAL.PAS	24.2	622	
MOVPARM.PAS	24.2	623	MOVParameter
MVPARMIN.PAS	24.2	626	VARParmINLINE
PATTERNS.PAS	22.4	498	
PIEMAN.PAS	22.5	512	
POLYGONS.PAS	22.5	502	
POWER.SRC	16.4	224	
PRINTET.PAS	19.3	314	
PTRINLN.PAS	24.2	631	PointerINLINE
PULL.SRC	16.12	232	
PULLDOWN.PAS	22.9	540	
PUSHPOP.PAS	14.3	185	
QUERYDSP.SRC	18.4	292	QueryAdapterType
QUIKSORT.SRC	14.4	187	QuickSort
RELAYON.SRC	20.4	367	
RELAYOFF.SRC	20.4	367	
ROLLEM.PAS	16.12	233	

Index

About the Author



Jeff Duntemann is founding editor of *TURBO TECHNIX, The Borland Language Journal.* Previously he was Technical Editor for *PC Tech Journal* and a Senior Programmer/Analyst for Xerox Corporation. A respected authority on Turbo Pascal, he wrote the best-selling First and Second Editions of *Complete Turbo Pascal,* and *Turbo Pascal Solutions,* all published by Scott, Foresman and Company.

>$24.95

THE DEFINITIVE GUIDE TO TURBO PASCAL 4.0 AND 5.0

Borland's Turbo Pascal 5.0 is here! The new compiler, along with Version 4.0, represent a total break with the popular Turbo Pascal 3.0. Greatly expanded and completely rewritten, the Third Edition of *Complete Turbo Pascal* becomes the authoritative practical programmer's tutorial and reference for both Turbo Pascal 4.0 and 5.0.

Whether you're a beginner or a programming wizard, you'll find lively, readily understandable guidance on programming the PC using Turbo Pascal Versions 4.0 and 5.0. All of the new features of 4.0 and 5.0 are covered, including:

- The new Large memory model
- The Integrated Development Environment and Integrated Debugging
- Separate compilation and overlays using Pascal units
- New low-level support for DOS and BIOS interface
- Hardware and software interrupt service routines in Pascal
- The Borland Graphics Interface (BGI)

No book yet published provides as much detail on such difficult topics as assembly language interface, program runtime internals, display adapter programming, and unit exit procedures. More than 8,500 lines of Pascal source code provide useful routines including pull-down graphics menus, interrupt-driven communication programs, joystick interface, directory searches, screen building, display adapter control, linked list management, and much more. Superbly written and heavily illustrated, this is the only guide to Turbo Pascal you'll ever need.

Jeff Duntemann is founding Editor of *Turbo Technix, The Borland Language Journal.* Previously he was Technical Editor for *PC Tech Journal* and a Senior Programmer/Analyst for Xerox Corporation. A respected authority on Turbo Pascal, he wrote the best-selling First and Second Editions of *Complete Turbo Pascal,* and *Turbo Pascal Solutions,* all published by Scott, Foresman & Company.

ISBN 0-673-38355-5

90000

9 780673 383556

£15.95

Scott, Foresman and Company

Version 5.0

Complete Turbo Pascal

Third Edition

Duntemann